Titre: Title:	A pattern-based framework architecture for distributed engineering applications
Auteur: Author:	Bin Chen
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Chen, B. (2004). A pattern-based framework architecture for distributed engineering applications [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7236/

UNIVERSITÉ DE MONTRÉAL

A PATTERN-BASED FRAMEWORK ARCHITECTURE FOR DISTRIBUTED ENGINEERING APPLICATIONS

BIN CHEN
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JANVIER 2004

Your file Votre référence
ISBN: 0-612-89189-5
Our file Notre référence
ISBN: 0-612-89189-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

A PATTERN-BASED FRAMEWORK ARCHITECTURE FOR DISTRIBUTED ENGINEERING APPLICATIONS

présenté par: CHEN BIN

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

M. GRANGER Louis, M.Sc.A., président

M. GUIBAULT François, Ph.D., membre et directeur de recherche

M. TRÉPANIER Jean-Yves, Ph.D., membre et codirecteur de recherche

M. DAGENAIS Michel, Ph.D., membre

To my lovely director and codirector

to whom I owe all my eternal thankfullness and obedience.

ACKNOWLEDGMENTS

Thanks very much to my director, Professor Francois Guibault, who gave me very important advices and excellent ideas when I met difficulties in my research, guided me to the right direction when I made mistakes in the project, and helped me to write the whole thesis both in English and French. I appreciate very much his efforts and contributions on my studying.

Thanks very much to my co-director, Professor Jean-Yves Trépanier who gave me good advices, spiritual and financial supports in the project. I also appreciate very much that he spent his precious time to read my thesis and gave me valuable suggestions.

Thanks very much to my mother-in-law Yongdi Wang, and my wife Wenlei Jin. They tried their best to do all the things when I was very busy in study and research.

Thanks to my colleagues, Qun Zhou, Daojun Liu, Yun Wang, Yuanli Wang, Xueyan Guo, Christophe Tribes, Djamel Bouhemhem and Amadou N'diaye who worked with me and brought me many happy days in CERCA and Ecole Polytechique.

RÉSUMÉ

Ce document présente l'architecture globale et les approches utilisées durant la conception de la plate-forme VADOR. Le but de la plate-forme VADOR est de permettre l'intégration transparente d'applications d'analyse en ingénierie dans un environnement de calcul distribué hétérogène. VADOR vise également à fournir, au sein d'une entreprise, des mécanismes de déploiement et de gestion des applications tant commerciales que développées à l'interne, qui permettent l'automatisation des processus de conception de produits et l'intégration d'algorithmes d'optimisation dans les phases de conception. VADOR a été développé en étroite collaboration avec la compagnie Bombardier Aerospace, qui a fourni la grande majorité des applications d'analyse, des processus à automatiser ainsi qu'un environnement industriel pour la validation du système.

Une architecture à plusieurs niveaux basée sur une approche client/serveur a été proposée pour la conception des principales composantes de la plate-forme. L'architecture comprend un client interactif en mode graphique agissant comme interface au système (Interface Usager Graphique ou IUG), qui permet la définition des données et le lancement des processus d'analyse, ainsi que des serveurs distincts pour le contrôle de l'exécution et la gestion des données, et des serveurs autonomes exécutables à distance pour l'encapsulation et l'exécution des applications patrimoniales.

Tant les données que les programmes d'analyse sont encapsulés dans une couche d'objets gérés au niveau de la plate-forme. Parmi les services fournis par cette couche d'objets, le contrôle de la propriété, le maintien des versions et le partage des composantes et des ressources de la plate-forme sont les plus importants pour une grande corporation telle que Bombardier.

Ce document se concentre principalement sur l'architecture de la plate-forme, présente le client servant d'IUG, les serveurs de gestion des données et d'exécution, la couche sous-jacente commune du modèle de données, et présente une discussion détaillée des patrons de conception utilisés pour la conception de la plate-forme VADOR.

ABSTRACT

This thesis presents the overall architecture and design of the VADOR application framework. The purpose of the VADOR framework is to enable the seamless integration of commercial and in-house analysis applications in a heterogeneous, distributed computing environment, and to allow the deployment of automatic design optimization algorithms based on the framework. VADOR is being developed in close collaboration with Bombardier Aerospace, who provides actual analysis applications, design processes in need of automation and test ground for the framework.

A multi-tiered client-server architecture has been proposed for the framework, which comprises a client GUI for interactive data definition and execution launching, separate data and execution servers, and autonomous remotely executable application wrappers. This thesis mainly focuses on the global framework architecture, introduces the client GUI, data and execution servers, the common underlying object model, and presents detailed discussions of the Design Pattern used in the Vador Framework.

Both analysis data and optimization algorithms are encapsulated in an object layer managed at the framework level. Among the services provided by this objects layer, ownership control, versioning and sharing of all framework components are among the most significant for a large corporation such as Bombardier.

CONDENSÉ EN FRANGAIS

Ce document présente l'architecture globale et les approches utilisées durant la conception de la plate-forme VADOR. Le but de la plate-forme VADOR est de permettre l'intégration transparente d'applications d'analyse en ingénierie dans un environnement de calcul distribué hétérogène. VADOR vise également à fournir, au sein d'une entreprise, des mécanismes de déploiement et de gestion des applications tant commerciales que développées à l'interne, qui permettent l'automatisation des processus de conception de produits et l'intégration d'algorithmes d'optimisation dans les phases de conception. VADOR a été développé en étroite collaboration avec la compagnie Bombardier Aerospace, qui a fourni la grande majorité des applications d'analyse, des processus à automatiser ainsi qu'un environnement industriel pour la validation du système.

Une architecture à plusieurs niveaux basée sur une approche client/serveur a été proposée pour la conception des principales composantes de la plate-forme. L'architecture comprend un client interactif en mode graphique agissant comme interface au système (Interface Usager Graphique ou IUG), qui permet la définition des données et le lancement des processus d'analyse, ainsi que des serveurs distincts pour le contrôle de l'exécution et la gestion des données, et des serveurs autonomes exécutables à distance pour l'encapsulation et l'exécution des applications patrimoniales.

Tant les données que les programmes d'analyse sont encapsulés dans une couche d'objets gérés au niveau de la plate-forme. Parmi les services fournis par cette couche d'objets, le contrôle de la propriété, le maintien des versions et le partage des composantes et des ressources de la plate-forme sont les plus importants pour une grande corporation telle que Bombardier.

Ce document se concentre principalement sur l'architecture de la plate-forme, présente le client servant d'IUG, les serveurs de gestion des données et d'exécution, la couche sous-jacente commune du modèle de données, et présente une discussion détaillée des patrons de conception utilisés pour la conception de la plate-forme VADOR.

Architecture globale de VADOR

L'architecture de la plate-forme VADOR est divisée selon les trois couches classiques de présentation, domaine d'application et données persistantes. Cette approche de conception permet de découpler l'architecture du système en de nombreux modules autonomes afin d'augmenter la flexibilité et permettre la réutilisabilité de chaque composante.

Les modules de VADOR comprennent:

1. Au niveau de la couche de présentation:
    - *Le VadorGUI*, qui procure une interface usager graphique permettant aux usagers de créer et de manipuler interactivement leurs propres *Data* et *Strategy Components*; ces composantes forment la base de l'information sur les données et les processus.
    - *Le DBExplorer*, qui est une application cliente qui fournit une interface graphique pour communiquer avec le *Librarian Server*, où les composantes créées dans le système sont entreposées. Le *DBExplorer* est un outil administratif, contrairement au *VadorGUI*, qui est un outil interactif s'adressant aux ingénieurs usagers du système.
2. Au niveau de la couche du domaine d'application:

- Le serveur *Librarian*, qui est responsable de la gestion des *DataComponents* et des *StrategyComponents*, et de l'interaction entre la plate-forme et le système de base de données.
- Le serveur *Executive*, qui est responsable de l'exécution des commandes lancées par les usagers au travers du *VadorGUI*, et de retourner les résultats d'exécution au *Librarian* lorsqu'une étape d'analyse est complétée.
- Les serveurs *Wrapper* et *Wrapper_servlet*, qui sont les interfaces distribuées des serveurs de ressources CPU. Ces serveurs sont contactés par le serveur *Executive*; ils créent les *DataComponents* et démarrent l'exécution des programmes d'analyse.
- Les programmes d'analyse, qui sont les applications patrimoniales encapsulées par la plate-forme.

3. Au niveau de la couche de données persistantes:
   - Les fichiers liés aux *DataComponents*, qui sont les fichiers servant à stocker les résultats d'analyse des applications d'ingénierie.
   - La base de données (*DBMS*), qui conserve la description de toutes les composantes directement gérées par la plate-forme, incluant la description des *DataComponents* et des *StrategyComponents*.

Patrons de conception dans VADOR

L'architecture du projet VADOR repose sur un modèle multi-niveau de type client/serveur, et sa conception a posé un grand nombre de défis. La majorité de ces défis ont été résolus à l'aide d'une approche de conception basée sur les patrons. Dans ce contexte, de nombreux patrons publiés et déjà bien documentés ont été utilisés,

dont les patrons *Composite*, *Strategy*, *Visitor*, *State*, *Proxy*, *Command*, *Template Method*, *Mediator*, *Adaptor* et *Observer*, afin de produire une architecture la plus extensible et facilement maintenable possible.

Une autre contribution majeure du présent travail réside dans le développement et l'évolution d'un nouveau patron de conception, nommé *Agent Actif* (*Active Agent* en anglais), basé sur les patrons *Active Object*, *Command*, *Proxy*, *Visitor* et *Strategy*. Ce patron tente de résoudre le problème de la concurrence dans un environnement de calcul distribué, et fonctionne selon le principe de l'agent mobile; cet agent regroupe la majorité des composantes de la plate-forme VADOR.

VADOR – Un système à base d'agents

L'ensemble de l'architecture du système VADOR repose sur l'idée d'agent, encapsulée dans le patron *Agent Actif*. Ce système agent est implanté au-dessus de la machine virtuelle de Java (Java Virtual Machine, ou JVM). Tant les serveurs que les clients sont exécutés sur la JVM. Ces programmes peuvent s'exécuter sur le même ou sur différents noeuds du réseau. Les agents s'exécutent sur les serveurs et interagissent avec les usagers du système au travers des applications de VADOR, et en particulier du VadorGUI. Les sous-sections qui suivent présentent respectivement la structure et l'applicabilité du patron Agent Actif, tel qu'il a été mis en oeuvre dans le système Vador.

Structure

La figure 1 montre le diagramme de collaboration du patron Agent Actif et identifie les participants au patron, qui coopèrent afin de fournir un service aux participants

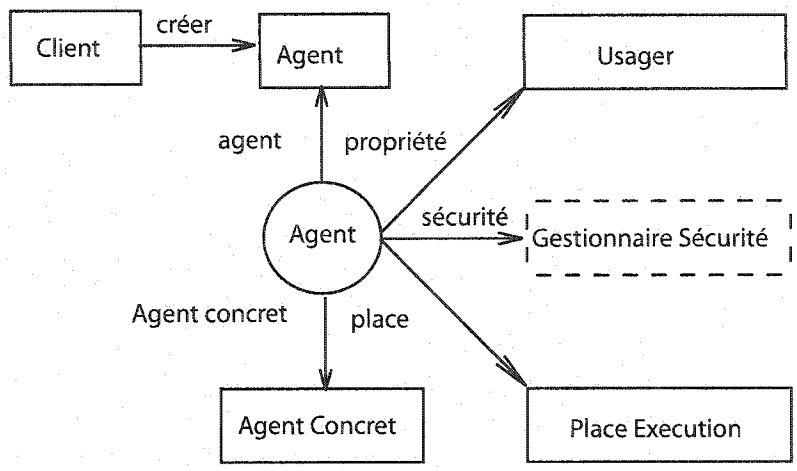

FIGURE 1 Structure du patron Agent Actif

externes. Voici une brève description de chaque participant, présenté en relation avec des instances réelles du patron dans le système VADOR.

- **Client**

  Le client crée et manipule les agents en utilisant les interfaces standards fournies par le patron. Dans le contexte du système VADOR, les clients représentent les applications qui jouent directement un rôle dans le système au nom d'un usager, comme l'application VadorGUI.

- **Usager**

  L'usager est identifié à l'aide d'un identificateur unique dans le système. Lorsque l'usager crée un agent à l'aide de l'une des applications de VADOR, son identificateur est inclu dans l'agent, et ce dernier devient son délégué dans le système.

- **Agent**

  La classe abstraite Agent est la partie visible et extensible du patron Agent

Actif. Cette classe définit le comportement abstrait des agents, qui inclut, dans le cas du système VADOR, une fonction `call` qui déclenche l'exécution de l'agent, et une fonction de garde `can_run` qui permet à chaque agent concret de définir ses propres conditions d'exécution.

- **AgentConcret**

  Les sous-classes AgentConcret(s) dérivent de la classe abstraite Agent et implantent le comportement défini par chaque agent afin de réaliser une tâche concrète. Par exemple l'agent `OpenStrategy` charge un objet de type `StrategyComponent` de la base de données et le transporte au niveau du serveur qui a fait la requète, alors que l'agent `SaveStrategy` permet de transférer un `StrategyComponent` vers le serveur `Librarian` et de le sauvegarder dans la base de données.

- **Gestionnaire de sécurité**

  Cette classe spécifie la politique de sécurité utilisée par les agents sous le contrôle du système Vador. Cette classe est planifiée, mais non implantée pour l'instant.

- **Endroit d'exécution**

  Cette classe définie l'environnement d'exécution des agents, qui correspond à la fois à l'endroit où l'agent est créé, et l'endroit où il réside en cours d'exécution. Au sein de la plate-forme Vador, les Endroits d'exécution sont les différents serveurs (`ExecutiveServer`, `LibrarianServer`, etc.).

Applicabilité

Le patron *Agent Actif* est utilisable de façon générale, lorsqu'un concepteur cherche à:

- Définir des objets actifs autonomes qui peuvent s'exécuter sur une plate-forme telle que le système Vador,
- Isoler les applications clientes des détails de bas niveau tels que les mécanismes de distribution ou les aspects de sécurité, tout en permettant aux clients d'obtenir des références à des agents et d'interagir avec ceux-ci de façon transparente,
- Simplifier le développement et la gestion d'applications dynamiques distribuées,
- Résoudre des problèmes de concurrence, de flexibilité et de croissance d'applications distribuées.

Les patrons de conception dans Vador

Outre l'utilisation du patron *Agent Actif*, qui a été utilisé pour concevoir l'architecture de base du système, Vador utilise un grand nombre d'autres patrons afin d'assurer une conception qui soit à la fois extensible, et facilement maintenable. Cette section présente l'utilisation des principaux patrons de conception qui se retrouvent dans Vador, classés en deux catégories, soit ceux qui se retrouvent dans la couche d'interface usager, et ceux qui ont été utilisés dans la couche liée au domaine d'application.

Les patrons de conception dans l'IUG

Le module VadorGUI se situe dans la couche de présentation, et son architecture utilise trois patrons importants: le patron *Mediator*, le patron *Template Method* et le patron *Adapter*.

- Le patron *Mediator* définit un object qui encapsule les mécanismes d'interaction entre un ensemble d'objets. Le médiateur favorise un couplage faible entre les objets en limitant les références maintenues par chaque objet vers les autres objets avec lesquels il doit interagir. Le médiateur permet par ailleurs de faire varier les mécanismes d'interaction entre les objets de façon indépendante des objets eux-mêmes.

  Dans le module VadorGUI, tous les menus doivent être configurés en fonction du type d'objet affiché dans la zone graphique de l'interface. Ce comportement collectif a été encapsulé de façon séparée dans un objet intermédiaire jouant le rôle de médiateur entre les Composantes et les menus. Cette approche de conception permet de localiser le comportement de configuration des menus dans l'objet médiateur, qui serait, sans cela, distribué parmi plusieurs objets. Apporter un changement au processus de reconfiguration des menus n'implique que de sous-classer le médiateur; les objets constituant le menu peuvent être réutilisés tels quels. Cette approche découple effectivement les menus des composantes et simplifie le protocole d'échange entre eux.

- Le patron *Template Method* définit le squelette d'un algorithme dans une opération, en déléguant la réalisation de certaines étapes aux sous-classes. Le patron *Template Method* permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans en changer la structure.

L'application VadorGUI fournit aux usagers une interface graphique pour manipuler des objets de type `DataComponent` et `StrategyComponent`. L'un des principaux objectifs de l'IUG est de présenter des vues spécialisées des différents objets qui sont manipulés à l'aide du système. À cette fin, le module de visualisation du modèle fournit plusieurs classes qui peuvent chacune reconfigurer la zone de visualisation selon les préférences de l'usager. Afin de créer et de gérer ces classes de visualisation, un ensemble fixe d'étapes sont utilisées, mais le comportement associé à chaque étape peut être différent pour chaque classe. La plupart des algorithmes de visualisation sont implantés sous la forme de méthodes *templates* selon le patron de conception du même nom. Il s'agit d'une technique fondamentale pour la réutilisation de code, qui mène à une inversion de la structure de contrôle, dans laquelle la classe parent appelle les méthodes des classes enfants plutôt que l'inverse. Cette inversion rend le code plus simple à comprendre, et plus robuste.

- Le patron *Adapter* vise à convertir l'interface d'une classe afin qu'elle se conforme aux attentes de clients. Le patron *Adapter* permet à des classes de travailler ensemble alors qu'elles ne pourraient pas le faire autrement.

  La plate-forme Vador fournit un outil de déverminage afin d'aider les usagers dans la localisation et la correction des problèmes rencontrés lors de l'exécution d'un `DataComponent`. Cet outil de déverminage conserve la trace de toute l'information d'exécution générée par la plate-forme et sauvegarde cette information dans un fichier XML. Lorsque l'usager doit procéder au diagnostique d'une erreur, l'outil affiche l'information d'exécution dans l'application VadorGUI. Étant donné que les données XML peuvent être interprétées à l'aide de noeuds DOM, qui forment en fait un arbre, la classe Java Swing `JTree` peut être utilisée pour afficher cette information. Cependant, comme l'interface d'un noeud DOM est différente de celle du `JTreeModel`, le

patron *Adapter* a été utilisé pour convertir d'une interface à l'autre.

Les patrons de conception dans la couche du domaine d'application

La couche associée au domaine d'application constitue le noyau fondamental du système Vador, et sa conception utilise sept patrons importants: les patrons *Command*, *Composite*, *Strategy*, *Visitor*, *State*, *Proxy* et *Observer*.

- Le patron *Command* encapsule une requête sous la forme d'un objet, permettant ainsi aux clients d'être paramétrés par différentes requêtes, queues ou demande de traces, et de supporter des opérations réversibles. La clef de ce patron tient à la définition d'une classe abstraite *Command*, qui déclare une interface commune pour exécuter toutes les opérations. Dans sa forme la plus simple, cette interface ne contient qu'une opération abstraite `Exécuter`. Les sous-classes concrètes de *Command* spécifient une opération en définissant la méthode `Exécuter`. Le récepteur de la requête ne requiert alors aucune connaissance supplémentaire, et se contente d'invoquer la méthode `Exécuter` pour déclencher l'opération.

  La plate-forme d'exécution du projet Vador présente une architecture composée de plusieurs couches de service. Le travail s'accomplit au sein du système en dépêchant des agents au travers du réseau sur les différents serveurs pour changer et gérer les données. Lorsqu'un agent arrive sur un serveur, il n'est pas nécessaire pour le serveur de connaître le type spécifique de l'agent. L'agent est plutôt conçu sur le modèle du patron *Command*, ce qui permet au serveur de déclencher son exécution sans en connaître les détails.

- Le patron *Composite* permet d'assembler des objets dans des structures afin de représenter des hiérarchies comprenant un tout formé de plusieurs parties.

Ce patron permet de traiter les objets et les compositions d'objets de façon uniforme.

Le système Vador accomplit du travail en manipulant et en changeant les objets `DataComponent` et `StrategyComponent`. Le projet Vador permet par ailleurs aux usagers de définir et d'exécuter des processus complexes en assemblant des processus plus simples et en les regroupant. Le patron *Composite* se prête parfaitement à cette représentation récursive des composantes, de façon à ce que les modules du système n'aient pas à distinguer entre les objets atomiques et composites. Partout où du code client est prêt à traiter un objet atomique, il est également en mesure de traiter un objet composite. Cette architecture permet également d'ajouter facilement des nouveaux types de données ou de processus.

- Le patron *Strategy* définit une famille d'algorithmes, chacun encapsulé individuellement dans une classe, ce qui les rend interchangeables. Le patron *Strategy* permet de faire varier les algorithmes de façon indépendante des clients qui les utilisent.

  Dans Vador, les différents processus d'analyse sont encapsulés dans des objets composites, qui doivent être exécutés de différentes façons. Certains processus doivent s'exécuter de façon séquentielle ou en parallèle, alors que d'autres doivent boucler et tester différents types de conditions avant de continuer ou non. Coder ces différents comportements directement dans les classes qui les utilisent est loin d'être désirable; ce problème est évité en définissant des classes pour chaque type d'exécution. Une méthode encapsulée de cette façon est une stratégie. Le patron *Strategy* fournit une hiérarchie de classes qui définit une famille d'algorithmes d'exécution, permettant de les interchanger à volonté, et rendant chacun plus facile à comprendre et à étendre.

- Le patron *Visitor* représente une opération qui peut être appliquée aux éléments d'une structure de données de façon externe. Le *Visitor* permet de définir des nouvelles opérations sans changer les classes des éléments sur lesquelles il opère.

  Le projet Vador représente les processus à l'aide d'objets `StrategyComponents`, qui forment un type d'arbre abstrait. Des opérations doivent être appliquées à chaque noeud de l'arbre, telles que "exécuter une stratégie" ou "sauver une stratégie dans la base de données". La majorité de ces opérations doit distinguer entre les différents noeuds constituant la `StrategyComponent`, ce qui exige de définir une opération de traitement différente pour chaque type de noeud. Idéalement, chaque opération devrait pouvoir être ajoutée de façon indépendante, et les différents type de noeuds devraient restés indépendants de ces opérations. Ces deux objectifs peuvent être atteints simultanément en définissant les méthodes associées à une opération dans une classe autonome, appelée `Visitor`, et à passer un objet de cette classe aux noeuds à mesure que l'arbre est traversé. C'est le noeud lui-même qui choisit alors la méthode appropriée pour réaliser l'opération en fonction de son propre type. Le patron *Visitor* rend facile l'ajout de nouvelles opérations, en regroupant les méthodes reliées à la réalisation d'une opération dans une classe autonome, ce qui augmente d'autant la flexibilité et l'extensibilité du système.

- Le patron *State* permet à un objet de modifier son comportement lorsque son état interne change. L'objet paraîtra alors avoir changé de classe.

  Lorsqu'un objet de type `StrategyComponent` se trouve sur le serveur `Executive`, nous utilisons le patron *Visitor* pour exécuter le processus qu'il décrit. L'exécution progresse alors d'un état à un autre, et l'exécution de l'état courant dépend de l'état précédent d'éxécution. Ce comportement est modélisé à l'aide du patron *State*. Ce patron localise les comportement liés à

un état spécifique, et permet de gèrer explicitement les transitions d'état en cours d'exécution.

- Le patron *Proxy* fournit un substitut ou un remplaçant à un autre objet dont on veut contrôler l'accès.

  Les serveurs Vador se trouvent dans la couche du domaine de l'application. Pour accéder aux serveurs, les clients ont recours à des objets *Proxy* (mandataires). Dans ce contexte, l'avantage d'utiliser un *Proxy* est de fournir un représentant local d'un serveur qui réside dans un espace d'adresse différent. Le client peut ainsi traiter avec le serveur distant comme s'il résidait dans son propre espace mémoire, et c'est le *Proxy* qui se charge d'établir la communication avec le serveur et de transmettre les requètes et les réponses. Cette approche de conception élimine la dépendance entre l'application cliente et les serveurs.

- Le patron *Observer* définit une relation un à plusieurs entre des objets de façon à ce que lorsqu'un objet change d'état, tous ses dépendants en soient avertis et mis à jour automatiquement. Les objets clefs dans ce patron sont le sujet (objet évènement) et le ou les observateurs. Un sujet peut accumuler autant d'objets observateurs dépendants que nécessaire. Tous les dépendants sont avertis lorsque l'état du sujet change.

  L'exécution d'une stratégie dans Vador peut engendrer plusieurs types d'événements, qui peuvent tous être potentiellement intéressants pour certains usagers. Par exemple, la création ou la modification de certains objets appartenant à un usager peut indiquer qu'un calcul est complété ou a été refait, ce qui peut entraîner une réponse d'un autre usager ou le déclenchement d'une nouvelle analyse. Lorsque de tels évènements se produisent, les usagers intéressés devraient être avertis automatiquement. Ce comportement

implique que les usagers dépendent de certains évènements ou objets, et devraient donc être avertis des changements qui se produisent dans leur état. De plus, il n'y a pas de limite au nombre d'usagers qui peuvent potentiellement dépendre de l'état d'un objet donné du système. Cette relation a été implantée à l'aide du patron *Observer*, qui laisse varier les évènements et les observateurs de façon indépendante. Les objets évènements peuvent être réutilisés sans que les observateurs le soient, et vice-versa. Ce patron permet donc d'ajouter des observateurs sans modifier ni les objets évènements ni les autres observateurs.

Utilisation du système Vador

L'utilisation du système Vador est relativement directe et conviviale. Le système fournit des outils pour aider les ingénieurs à intégrer leurs applications d'analyse, à définir les processus, et à déployer des algorithmes automatiques d'optimisation basés sur le système.

Voici un bref exemple de l'utilisation de Vador pour accomplir une tâche d'optimisation et gérer les résultats d'analyse produits.

Un exemple de processus d'analyse utilisant Vador

Cet exemple illustre plusieurs aspects de l'utilisation du système Vador pour l'implantation d'un processus d'optimisation de profils d'ailes d'avion. Cet exemple se penche essentiellement sur l'intégration du processus dans Vador sous la forme d'objets, et s'étend très peu sur les détails de l'application d'analyse elle-même. Brièvement, le processus cherche à construire une courbe NURBS ayant un nombre

minimum de points de contrôle afin d'approximer un profil d'aile d'avion représenté initialement de façon discrète par des points de mesure.

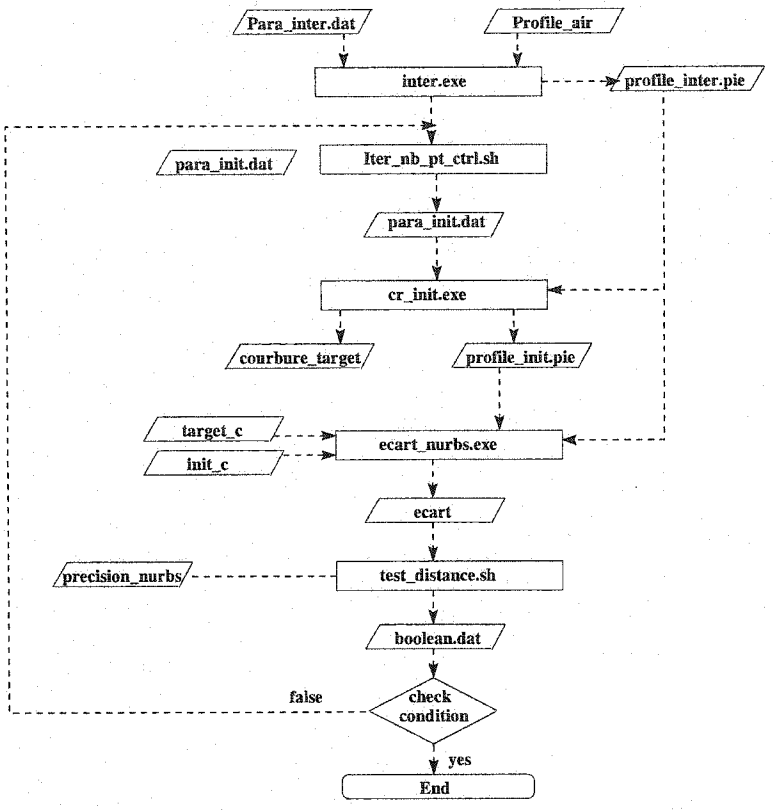

FIGURE 2 Processus d'optimisation d'un profil d'aile

La figure 2 présente le processus d'optimisation sous la forme d'un organigramme. Ce processus se divise en deux phases:

1. **La phase initiale** crée la courbe cible continue à partir de la représentation discrète, et de certains paramètres de contrôle.

   Pour ce faire, l'application `inter.exe` reçoit deux paramètres en entrée – `Para_inter.dat` et `Profile_air` qui fournissent les données nécessaire à la création de la courbe cible. Comme ces fichiers sont tous les deux créés à

l'extérieur du processus, on nomme ceux-ci données d'entrée externes (*external inputs*).

2. **La phase d'optimisation** crée successivement une série de courbes d'approximation, optimise la position et le poids de chacun des points de contrôle de la courbe, et compare les distances maximum et moyenne de la courbe d'approximation obtenue avec la courbe cible. Si l'erreur d'approximation est plus petite que l'erreur cible fixée pour le processus, la boucle se termine. Sinon, la boucle se poursuit, en incrémentant de le nombre de points.

   Cette phase se divise ainsi en plusieurs étapes:

   (a) Calcul du nombre de points,

   (b) Initialisation des nouveaux points de contrôle,

   (c) Calcul de la distance entre la courbe cible et la courbe d'approximation,

   (d) Vérification du critère de distance et du nombre d'itérations maximum.

Voici les principales étapes nécessaires pour définir les objets pertinents dans Vador:

1. Utilisation de l'application VadorGUI

   Cette application permet aux usagers du système Vador de défnir et de manipuler interactivement leurs propres objets `DataComponent` et `StrategyComponent`;

2. Définir les objets de type `DCType`. Le système Vador utilise des objets de type `DCType` pour décrire les types de données produites par chaque processus. Des types de données différents utilisent des objets `DCType` différents. Par exemple, l'objet `profile_inter` décrit un type de données utilisé en entrée par le programme `inter.exe`, dont le fichier de sortie porte l'extension ".pie".

3. Définir les objets de type `StrategyComponent`

   Le système Vador utilise des objets de type `StrategyComponent` pour décrire les processus d'analyse.

   Le processus en exemple peut être divisé en deux processus exécutés en séquence. Le second processus est un processus en boucle qui inclut plusieurs autres sous-processus, qui sont tous exécutés en séquence. Nous retrouvons donc le patron *Composite* dans la définition du processus global en deux étapes, et du processus de la seconde étape, qui en comprend quatre autres. Chaque processus utilise une stratégie qui lui permet soit d'exécuter un programme réel, soit de contrôler l'exécution des sous-processus qu'il contient. Par exemple, la stratégie `SequentialSgy` est utilisée pour définir les aggrégations de sous-processus exécutés séquentiellement, et la stratégie `DoWhileSgy` est utilisée pour définir le processus en boucle. Chaque programme, tels que `inter.exe` et `Iter_nb_pt_ctrl.sh`, est encapsulé dans un processus individuel qui utilise la stratégie `AtomicSgy` pour son exécution.

4. Définir les objets de type `DCInstance`

   Le projet Vador utilise des objets du type `DCInstance` pour décrire les exécutions concrètes d'un processus.

   Une fois que le processus a été défini à l'aide des objets de type `DCType` et `StrategyComponent`, l'usager est prêt à exécuter le processus. Pour ce faire, l'usager doit indiquer au système qu'il désire créer une instance concrète d'un type de données qu'il a préalablement défini comme résultat d'un processus, et indiquer, au besoin, quel processus doit être utilisé pour construire cette instance. En cours d'exécution, le système Vador va instancier des objets de type `DCInstance` pour chaque paramètre transmis à une stratégie et chaque donnée concrète qui est produite. Ces objets encapsulent les résultats pro-

duits par les programmes d'analyse.

5. Sauvegarder les objets de Vador dans la base de données

   Le système Vador à recours à un serveur spécialisé, le serveur *Librarian*, pour manipuler directement la base de données, et y stocker les données décrivant les instances produites en cours d'exécution des processus.

6. Exécuter le processus

   Le système Vador utilise un autre serveur spécialisé, le serveur *Executive*, pour gérer l'exécution des processus. C'est le serveur *Executive* qui est responsable de coordonner les différentes phases de l'exécution d'un processus complexe, d'envoyer les données produites au serveur *Librarian* pour qu'elles soient stockées dans la base de données et de retourner les résultats ou les erreurs au client.

Résultat d'une exécution typique

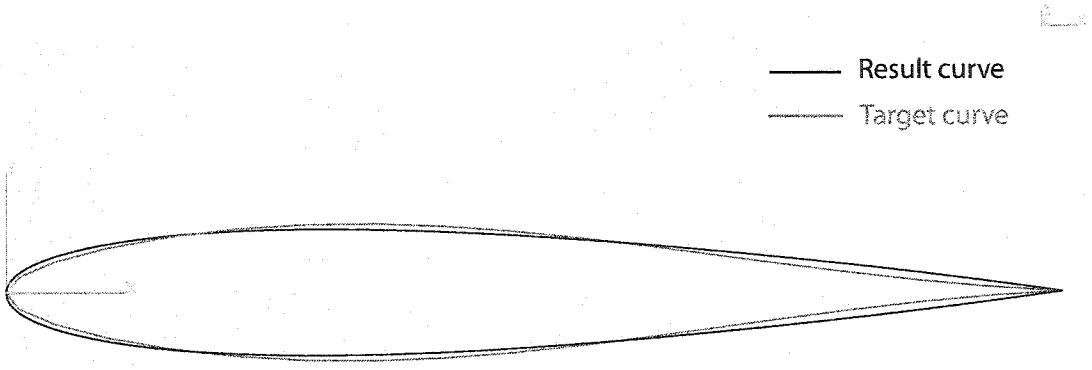

FIGURE 3 Résultat d'une optimisation d'un profil d'aile

La figure 3 montre le résultat obtenu lors de l'exécution du processus d'optimisation

décrit à la section précédente. La courbe en noir est la courbe optimale obtenue (dont la description est stockée dans le fichier `profile_init.pie`), alors que la courbe en gris est la cible (dont la description est contenue dans le fichier `profile_inter.pie`). Étant donné que la valeur de l'erreur admissible a été choisie très grande, on observe une bonne distance entre les deux courbes. Il faut cependant noter que dans des cas réels d'utilisation, les valeurs d'erreur choisies font en sorte que les deux courbes sont absolument superposées.

Extensions au système Vador

L'utilisation des patrons de conception permet d'étendre facilement le système Vador. Plusieurs points d'extension ont été prévus dans l'architecture du système, en fonction des besoins appréhendés de développement du système. Évidemment, ces extensions sont possibles sans exiger de modification à l'architecture globale du système. Les principaux points d'extension de Vador peuvent être classés en trois catégories:

1. Les extensions au niveau de l'IUG

   Ces extensions comprennent entre autres:

   - La possibilité d'ajouter de nouvelles capacités d'affichage et de configuration des menus pour les objets affichés, à l'aide du patron *Mediator*,
   - La possibilité de définir de nouveaux types de vues pour les `DataComponents` et `StrategyComponents` à l'aide du patron *Template Method*.

2. Les extensions sous la forme de nouveaux clients

   Bien que normalement, les usagers du système accèdent à la fonctionnalité de Vador au travers de l'application VadorGUI, tous les mécanismes et interfaces

nécessaires sont en place pour permettre d'ajouter d'autres types de clients aux différents serveurs de Vador. Cette possibilité a d'ailleurs été utilisée pour des travaux de recherche sur les algorithmes de répartition de charge utilisant Vador, (voir Liu 2003), qui utilisaient un client en lot pour déclencher l'exécution de tâches.

3. Les extensions dans la couche liée au domaine d'application. Plusieurs points d'extension ont été prévus dans la couche du domaine d'application, dont:

   - L'ajout de nouveaux agents, basés sur le patron *Command*,
   - L'ajout de nouveaux types d'objets composites, tant pour les stratégies que pour les types de données, selon le patron *Composite*,
   - L'ajout de nouvelles stratégies, selon le patron *Strategy*,
   - L'ajout de nouvelles opérations sur les stratégies, à l'aide des patrons *Visitor* et *State*.
   - L'ajout de nouveaux types de serveurs, et leur intégration dans le système à l'aide du patron *Proxy*.

Bilan de l'utilisation des points d'extension dans Vador

Nous avons procédé à une étude de faisabilité afin d'évaluer les possibilités d'extension du système Vador. L'objectif de cette étude était de vérifier l'impact et la faisabilité de l'ajout d'un nouveau type de client et de serveur basé sur les protocoles Web pour gérer les données et les processus d'analyse, et consulter les résultats d'exécution des processus sous le contrôle de Vador.

Cette étude a permi d'identifier l'ensemble des classes à ajouter au système Vador, et de valider le potentiel d'intégration de ces classes dans le système existant. À

la lumière de cette étude, trois conclusions importantes peuvent être tirées quant à l'architecture globale du système:

1. Le système Vador fournit tous les points d'extension nécessaires à l'ajout d'un nouveau client et d'un nouveau serveur, ce qui permet d'ajouter de nouveau types de services au système sans nécessité de modifier l'architecture proposée,

2. Les points d'extension de Vador sont facilement identifiables et bien documentés, dans un format explicite utilisant un langage familier des développeurs. La figure 4 présente les points d'extension liés à la définition de l'Agent uti-

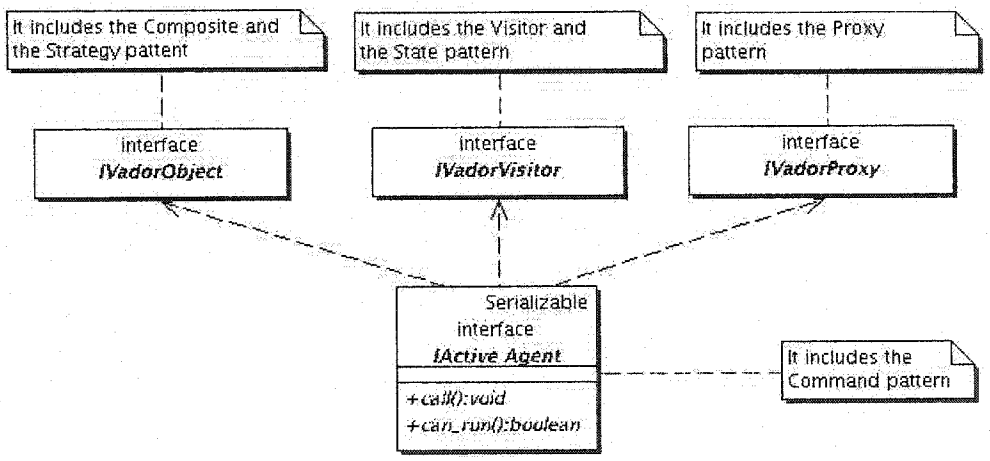

FIGURE 4 Diagramme des points d'extension d'un Agent

lisé dans Vador. Lorsqu'un nouveau service est ajouté au système, toutes les interfaces documentées dans ce diagramme doivent être implantées. Il est possible d'utiliser les comportements de défaut proposés par le système pour une bonne proportion des méthodes.

3. Le déploiement d'un nouveau service est simplifié.

Lorsqu'un nouveau service basé sur les agents doit être déployé dans Vador, les serveurs existants n'ont pas à être modifiés, ni même redémarrés, étant donné que le mode d'interaction d'un serveur avec un agent est déjà connu.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

ASS	Agent Support System
CERCA	CEntre de Calcul en Recherche Appliqué
CFD	Computational fluid dynamics
CORBA	Common Object Request Broker Architecture
CSA	Client/Server Architecture
CSM	Computational structural mechanics
C/S	Client/Server
DBMS	Database Management System
EJB	Enterprise Java Bean
GUI	Graphic User Interface
HTTP	Hypertext Transport Protocol
JVM	Java Virtual Machine
MA	Mobile Agent
MAS	Multi-Agent System
MDO	Multidisciplinary Design Optimization
MVC	Model View Controller
OMG	Object Management Group
OSF/DEC	Open Software Foundation / Distributed Computing Environment
RDA	Remote Data Access
RPC	Remote Procedure Call

SQL	Structured Query Language
URL	Uniform Resource Locater
VADOR	Virtual Aircraft Design and Optimization fRamework
WWW	World Wide Web

Caractères usuels

D	P/π
P	$\pi * D$

Caractères grecs

π	P/D

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Preliminary Engineering Design

Computational based design is a rapidly evolving field. Mature technologies for the simulation and analysis of complex physical systems are now available in most engineering disciplines. For example, in aeronautical applications, the aerodynamics and structures disciplines each propose their own state-of-the-art methods for respectively simulating the behavior of fluid flow and of the stress and strain of structures: high-fidelity computer codes now solve 3D Navier-Stokes equations for a full aircraft in computational fluid dynamics (CFD) and complete finite-element model for a full aircraft in computational structural mechanics (CSM).

In parallel to the development of these state-of-the-art high fidelity simulation models, design methods have been developed to help engineers take better design decisions under constraint. Many proposed design methodologies are based on the formulation of an appropriate optimization problem where analysis tools are used to compute the cost function. These optimization methods have developed in parallel with the analysis tools since the 60's in CSM and the 70's in CFD, although they were used mostly with simple physical modeling having low-CPU requirements.

In spite of this fast evolution of isolated disciplines, only timid efforts have been targeted toward design optimization as a whole, for example to minimize the Direct Operational Cost or maximize the Return on Investment for airline companies. To some extent, this type of optimization has been implemented using

crude representations of disciplines and applied, for instance, during conceptual design in aeronautics. However, as high fidelity models from each discipline mature, there is now an incentive to use these better models in a more global design environment; this has given rise to the development of Multidisciplinary Design Optimization (MDO) (Sobieszczanski-Sobieski and Haftka (1996),Sobieszczanski-Sobieski and Haftka (1997)) as a distinct discipline. MDO's primary objective is to develop methodologies and tools to tackle the formidable challenges of integrating high-fidelity physical models in a design environment and allow the synergism of mutually interacting disciplines to be fully exploited.

1.2 The Vador Project

In this context, and in close collaboration with Bombardier Aerospace, a group of research at Polytechnique and CERCA is currently working on a project, called Virtual Aircraft Design and Optimization fRamework (VADOR), to develop a framework for integrating multi-disciplinary and multi-fidelity engineering analysis programs.

The main goal of this project is to provide technical engineers with an integration solution of legacy analysis applications and the associated data management tools necessary to handle the vast amounts of result data produced daily by the various departments of an aerospace organization. Most legacy applications are custom made in-house applications which need specialized interfaces to communicate with one another. These applications often carry a high cost in terms of use and interfacing time on the part of the engineers trying to get them to communicate in new ways with each other. The Vador project aims to significantly reduce interfacing time through an effective documentation of available communication paths among

applications, and by providing access to centralized analysis process descriptions that engineers can use or adapt to new needs.

One of the main challenges in applying MDO methodology in a large aeronautical corporation lies mainly in organizational aspects of engineering design and analysis. Analysis process documentation and result data access are central to an effective implementation of MDO, and an organization-wide approach to the management of these issues must be enforced. The Vador project, which aims to provide the necessary foundations to implement the MDO methodology within Bombardier is thus not targeted toward one technical department or application, but rather spans the full spectrum of technical skills and processes involved in aeronautic design.

1.3 Key Components of the Vador Framework

- **Distributed system**

  The Vador Framework enables the seamless integration of commercial and in-house analysis applications in a heterogeneous, distributed computing environment, and allows the deployment of automatic design optimization algorithms based on the framework. Object-oriented methodologies are used in the development of the Vador Framework, with the implementation being done using the Java programming language.

  Indeed, an effective framework enabling a multidisciplinary design optimization(MDO) should provide users with a flexible and configurable data model which can adequately satisfy the evolving requirements of engineers using computational-based analysis-and-design programs. Such a system must provide capabilities for the automation and integration of various processes used by engineers, regardless of the platform these applications run on, and it

must also support and promote collaboration and data sharing in a platform independent manner. These requirements are at the root of the choice of a Java based distributed system.

- **Manipulation of the user data in its native format**

  The Vador Framework treats all the data as objects. Design-and-analysis data, contained in actual data files is encapsulated in an object named DataComponent. The DataComponent object contains an appropriate set of attributes required for data management, but leaves the engineer's data itself in the files that are being encapsulated.

- **Encapsulation of engineering applications**

  The Vador Framework also treats engineering applications as objects. These objects named StrategyComponent(s), encapsulate the design-and-analysis methodologies or processes. They represent the basic methods and the work and data flows required to transform data in a given process. The StrategyComponent can include user programs which can create the data files encapsulated in the DataComponent. The programs are usually executable legacy programs to be executed on a specific machine on the network.

- **Graphical user interface**

  The Vador Framework offers a graphical user interface which is the visual part of the Java program that runs on the user's machine. The users create and manipulate interactively their own DataComponent and StrategyComponent through this graphical user interface application.

- **Saving data in the database**

  Once created, the DataComponent and StrategyComponent objects are saved in the database. The present architectural design supports the separation of

the basic data, usually contained in files and potentially rather large, from the descriptive information. Only the descriptive information, or meta-data, is stored in the Database.

1.4 Vador and Design Patterns

Among the wide range of issues involved in the design of the Vador framework, architectural issues rank among the most prominent in terms of risk mainly because of the major impact of the architecture on the extensibility and maintainability of the application. A very recent approach to architectural design involves heavy reliance upon design patterns and pattern languages to realize distributed frameworks and improve their performance.

Since the Vador project is a multi-tiered client-server architecture framework, there are many challenges that must be resolved and design patterns have been proved useful to help capture and reuse the static and dynamic structure and collaborations of key participants in a software design.

This document will focus on architectural aspects of the Vador framework and the use of design patterns to solve fundamental maintenance, evolution, distribution and concurrency problems encountered in the design and realization of the Vador framework. In the Vador project, we have used numerous previously published design patterns including the Composite, Strategy, Visitor, State, Proxy, Command, Template Method, Mediator, Adapter, and Observer patterns, in order to achieve a truly extensible and maintainable architecture. The use of design patterns in the context of distributed software architectures is still a relatively recent topic for which research is very active. In the case of implementing the Vador framework, great care has been taken to propose an architectural design of the framework

which is both scalable and extensible. This resulting reusable framework architecture constitutes one of the chief contributions of this work.

Another major contribution of the current work lies in the evolution, specialization and implementation of a new design pattern, named Active Agent pattern for the core development of the framework. This pattern is based on the Active object, Command, Proxy, Visitor and Strategy patterns. This pattern tries to resolve the concurrency problem in the distributed framework. It works as a mobile agent and includes almost all the components in the Vador framework

This document is structured in the following way:

- chapter 2 discusses the review about design patterns, mobile agents and their relationship with distributed systems.
- chapter 3 introduces the global architectural design of the Vador framework.
- chapter 4 & 5 presents the pattern based architecture of the Vador framework.
- chapter 6 uses a real example to illustrate how the Vador framework works.
- chapter 7 gives a case study to show how easy it is to extend the Vador system.
- chapter 8 concludes this document and discusses future work.

CHAPTER 2

REVIEW OF LITERATURE

2.1 Design Pattern

2.1.1 What is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander *et al.* (1977)]. This excellent description by Alexander, which applies to the patterns in buildings and towns, can also be applied to the object-oriented design patterns. Design patterns in software engineering are expressed in terms of objects and interfaces instead of walls and doors, but at the core these two kinds of patterns solve problems in similar contexts.

One of the most famous framework was the Model-View-Controller (MVC) framework for Smalltalk [Krasner and S.T. (1988)] which divided the user interface design problem into three parts. Among these three parts, one contains the computational aspects of the program and is called the model layer, a second part presented the user interface and is called the view, and the third part contains control aspects of the application, which interacted between the user and the view.

The purpose of this structure is to separate objects among the different parts of an application, with each part having its own rules for managing data. The proposed structure also controls the communication between the user, the GUI and the

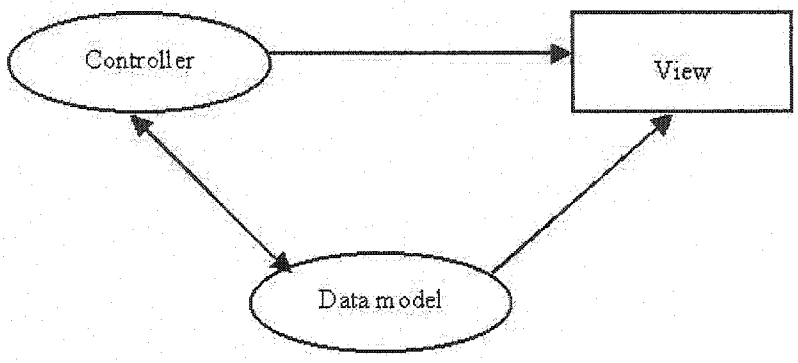

FIGURE 2.1 MVC

data, and it carefully separates the responsibilities among the parts. The objects in the three parts talk to each other using a restrained set of connections, which are implemented as a set of a few predetermined communication channels. The MVC framework is an example of a powerful reusable framework that uses design patterns.

In other words, design patterns describe how to establish communication between the objects while hiding their data models and methods from each other. Keeping this separation has always been an objective of good object oriented programming.

Some useful definitions of design patterns have emerged as the literature in the field has expanded (cited by Cooper (1998)):

- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development."(Pree, 1994)
- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed designed... and implementation."(Coplien & Schmidt, 1995).

- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it"(Buschmann, et.al.1996)
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components."(Gamma, et al.1993)

Although it is helpful to draw analogies to architecture, cabinet making and logic, design patterns are focused on describing how to design objects and how to communicate between these objects. In fact, sometimes one can think of them as communication patterns. The design of simple, but elegant, methods of communication makes many design patterns very important.

Design patterns can exist at many levels from very low level specific solutions to very high level general solutions. In general, a pattern has four essential elements:

1. The **pattern name** is a handle that can be used to describe a design problem, its solutions, and consequences in a word or two. Giving a name to the pattern increases collective design vocabulary. It lets designers describe solutions at a higher abstraction level. Using a vocabulary for patterns enables designers to talk about patterns with their colleagues, in their documentation, and even to themselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names is one of the hardest tasks in developing a Design Pattern catalog.
2. The **problem** describes when to apply the pattern. It explains the problem and its context. It describes specific design problems such as how to represent algorithms as objects. It describes also class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include many conditions that must be met before applying the design pattern.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. As the design pattern is like a template that can be applied in many different situations, it doesn't describe a particular concrete design or implementation, it describes an abstract solution of a design problem, and describes also how to arrange generally the elements (classes and objects) to solve the problem.

4. The **consequences** are the results and trade-offs of applying the design pattern. Although consequences are often unvoiced when design decisions are described, they are very important for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

Key motivations for people to document patterns include the following points (As mentioned in Schmidt *et al.* (1996)):

1. Success is more important than novelty.

   A pattern becomes valuable only if it has been used successfully in several occasions. Because new techniques are often untested, novelty can be a liability. Finding a pattern is an act of discovery from experience, not invention. We can describe a new technique as a design pattern, but we can only know its value after it has been tried. This is why most patterns describe several uses.

2. Emphasis on writing and clarity of communication.

   Most design patterns documentation describes the solution in a standard format. People hope one day they can have handbooks for software engineers. Therefore, they write their patterns in a form of catalog entry. That means pattern descriptions are a literary style and technical documentation.

It is very important to give a clear description of design patterns, as they stem from people's collective experience developing complex software systems. In many cases, the reason of the failure of a project originates in developers inhability to communicate good software designs, architectures, and programming practices to each other. The pattern descriptions that are well written help improve communication by naming and concisely articulating the structure and behavior of solutions to common software problems.

3. Qualitative validation of knowledge.

   Design patterns can help describe concrete solutions to software problems, instead of quantifying or theorizing about them. People also do the theoretical and quantitative work, but they feel such works are better in an another context, not included in discovery and documentation of patterns. The goal is to use the creative process that expert developers use to build high quality software systems.

4. Good patterns arise from practical experience.

   Every experienced developer has valuable patterns that people would like to share. The experience of all software developers should be valued, documenting design patterns and framework architectures is a concrete way of documenting designers' experience.

5. Recognize the importance of human dimensions in software development.

   The design patterns can not and should not replace developer creativity by imposing rigid design rules. They can not replace programmers with automated CASE tools. Instead, studying design patterns is a way to recognize the importance of human factors in developing software. This recognition appears in design patterns when they discuss their effect on the complexity and

understandability of software systems. In addition, this recognition shows itself in patterns on effective software process and organization.

2.1.2 Defining a Pattern language

When we put the related patterns together, we get a "language" – Pattern language, this language provides a process for the orderly resolution of software development problems. Pattern languages are not formal languages, they are collections of interrelated patterns, they provide a vocabulary for talking about a particular problem. These languages help to both:

- Define a vocabulary for talking about software development problems.
- Provide a process for the orderly resolution of these problems.

2.2 Object-Oriented Application Frameworks and design patterns

Over the past ten years, computing power and network bandwidth have increased noticeably. Billions of interactive and embedded computing and communication devices are in operation throughout the world. These powerful computers and networks are available largely at commodity prices, built with robust common-off-the-shelf (COTS) components, and will inter-operate over an increasingly convergent and pervasive Internet infrastructure.

However, software design and implementation of system efficiently using this hardware infrastructure are still expensive and error-prone. There are much of the cost and effort that comes from the continuous re-discovery and re-invention of core concepts and components in the software industry. In particular, it is hard to build

correct, portable, efficient, and inexpensive applications from scratch, one reason being in the growth of many different sorts of hardware architectures, operating system and communication platforms.

Object-oriented application frameworks "are a good technology for reifying proved software designs and implementations which can reduce the cost and improve the quality of software. It is a reusable, 'semi-complete' application that can be specialized to produce custom applications" [Johnson and Foote (1988)].

The most benefits of small object-oriented application frameworks come from "the modularity, re-usability, extensibility, and inversion of control they provide to developers" [Fayad and Schmidt (1997)].

Frameworks are closely related to small patterns, some of the most useful patterns describe frameworks. People can view these patterns as abstract descriptions of frameworks that enable widespread reuse of software architecture. Similarly, frameworks can be viewed as concrete implementations of patterns that makes it easy to directly reuse design and code. The difference between patterns and frameworks is that patterns describe the solution in language-independent manner, and the frameworks are generally implemented in a particular language. Design patterns and frameworks are highly systematic concepts, with neither subordinate to the other. We can expect that the next generation of object-oriented frameworks will explicitly include many patterns and patterns will be widely used to document the form and contents of frameworks.

2.3 Patterns for Concurrent, Parallel, and Distributed Systems

In theory, system performance, reliability, scalability, and cost-effectiveness all can be improved by developing software application that uses concurrent and networked services. In practice, as there are many differences between stand-alone and networked application architectures, it is hard to develop efficient, robust, extensible, and affordable concurrent and networked applications.

In stand-alone application architectures, user interfaces, application service processing, and persistent data resources reside within one computer, with peripherals attached directly to it. In contrast, in networked application architectures, interactive presentation, application service processing, and data resources may reside in multiple different host computers and services connected together by local area or wide area networks. The following definitions are taken from Schmidt *et al.* (1999).

Concurrency is about a family of policies and mechanisms that enable one or more threads or processes to execute their service processing tasks simultaneously.

A **Process** is a collection of resources, such as virtual memory, I/O handles, and signal handlers, that provide the context for executing program instructions.

A **Thread** is a single sequence of instruction steps executed in the context of a process. In addition to an instruction pointer, a thread consists of resources, such as a run-time stack of function activation records, a set of registers, and thread-specific data.

The use of single-threaded processes makes it easy to develop certain types of concurrent applications, such as remote logins, because the separate processes could not interfere with each other without explicit programmer intervention. But it is hard to use single-threaded processes to develop networked applications.

Modern operating systems provide multi-threaded concurrency mechanisms to overcome the limitations of single-threaded processes, these mechanisms support the creation of multiple processes, each of the processes may contain multiple concurrent threads.

The threaded programming models give us four benefits:

1. They improve performance transparently by using the parallel processing capabilities of hardware and software platforms.
2. They improve performance explicitly by allowing programmers to overlap computation and communication service processing.
3. They improve perceived response time for interactive applications, such as graphical user interfaces, by associating separate threads with different service processing tasks in an application.
4. They simplify application design by allowing multiple service processing tasks to run independently using synchronous programming abstractions, for example two-way method invocations.

Networked and concurrent applications provide many benefits, but they are harder to design, implement, debug, optimize, and manage than stand-alone applications. For example, developers must address topics that are normally non-relevant or less problematic for stand-alone applications in order to handle the requirements of networked applications.

General-purpose design patterns, such as Adapter and Wrapper Facade can be used to shield concurrent software from complexities. In addition, there is off-the-shelf 'infrastructure' middleware, such as ACE and JVMs, that are now widely

available and that implement some patterns into efficient and reusable object-oriented operating system encapsulation layers.

But many challenges still remain. What is needed is for developers to learn existing successful patterns for developing concurrent applications, and to evolve design methods into new patterns and frameworks as they prove successful in new components, frameworks, and system architectures.

2.4 Mobile agent and Distributed Systems

Another important thread of thought in the modern software community to deal with distributed software is to rely on code mobility. This has recently lead to the appearance of a large number of mobile agent platforms. A mobile agent is a small intelligent program that moves through a network automatically, searching for and interacting with services on the user behalf. These systems use specialized servers that interpret the agent's behavior and communicate with other servers. A Mobile Agent has inherent navigational autonomy and can ask to be sent to some other nodes.

Mobile Agents should be able to execute on every machine in a network and agent code should not be needed to be installed on every machine that the agent could visit. Therefore Mobile Agents use mobile code systems such as Java and the Java virtual machine, which enable them to load classes at runtime over the network.

A brief comparison with traditional network architectures helps to better understand mobile agents and their behavior. Figure 2.2 illustrates the network behavior of a typical client/server application.

A client/server application typically has two pieces: a client piece and a server piece.

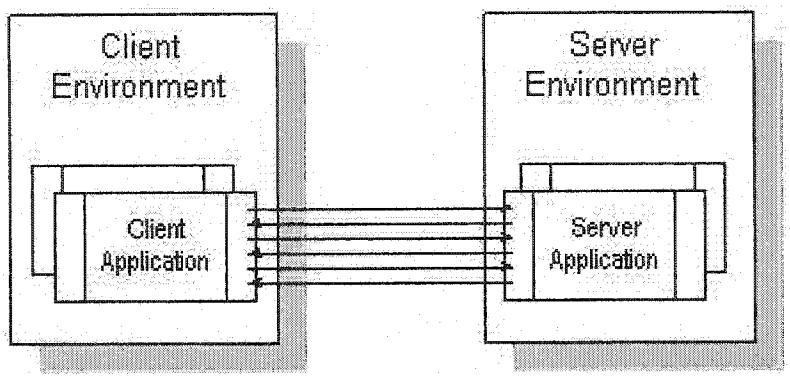

FIGURE 2.2 The typical client/server application communicates by using requests and responses system, which require a round trip across the network.

Normally, the client pieces and server pieces reside on separate machines and they communicate with each other through a common network. When the client needs data of one special server or access to resources that this server provides, the client sends a request to the server through the network. The server sends a response to the request. This "handshake" occurs again and again in a traditional client/server architecture. Each request/response cycle needs a complete round trip across the network.

Comparing with the client/server architecture. Figure 2.3 describes the mobile agent architecture .

As in the client/server architecture, there are a client piece and a server piece. The difference between them is how they communicate. When the client in the mobile agent architecture needs data on a special server or access to a resources that the server provides, the client doesn't talk to the server through the network. In fact, the client actually migrates to the server's machine. After arriving on the server, the client communicates with the server locally. When the entire transaction is

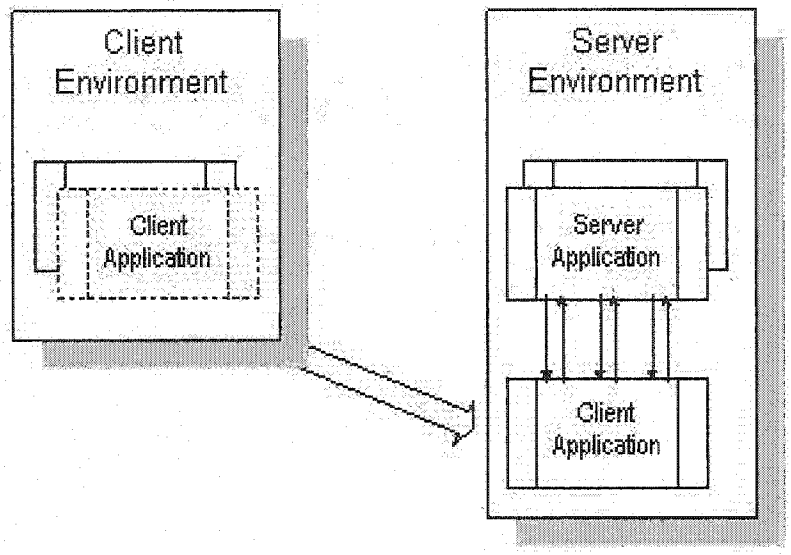

FIGURE 2.3 In the mobile agent architecture, the client actually migrates to the server to make a request directly, rather than over the network

completed, the mobile agent returns home with the results.

Mobile agents try to solve the problem associated with limited client/server network bandwidth. Network bandwidth in a distributed application is an important resource. A transaction or query between a client and the server may require many round trips over the network to complete. Each trip creates network traffic and consumes bandwidth. If there are many clients and/or many transactions in a system, the total bandwidth requirements may exceed available bandwidth, and lead to poor performance for the application. By creating an agent to handle the query or transaction, and sending the agent from the client to the server, network bandwidth consumption is reduced. So instead of completing a request by passing several times over the network, only sending and returning the agent is needed.

Agent architectures also solve the problems created by intermittent or unreliable

network connections. In most network applications, the network connection must be kept during the entire time of a transaction of a query. If the connection goes down, the client must restart the transaction or query from the beginning. Agent technology allows a client to dispatch an agent handling a transaction or query into the network when the network connection is alive. The client can then go off-line. The agent will handle the transaction or query automatically, and bring the result back to the client when it re-establishes the connection.

2.5 The Agent Pattern overview

Patterns have been proved extremely useful to the object-oriented programming community. As for the development of agent-based designs and architectures, agent patterns are beginning to appear in the Agent community. The Agent Pattern provides a clean and easy approach to develop agent-based system. This section presents some of the attempts to structure agent systems as pattern-based frameworks:

- **AgentSpace**

  INESC & IST Technical University has developed an Agent based system called **AgentSpace**, its Agent pattern encapsulates a business specific class (a specialization of the Agent class), with some user identification and a specific security policy, providing distribution, security and persistence transparency.

  As described in Silva and Delgado (2001), the Agent pattern forms a pattern language whose main components are presented in Figure 2.4:

  The main participants to the Agent Pattern are:

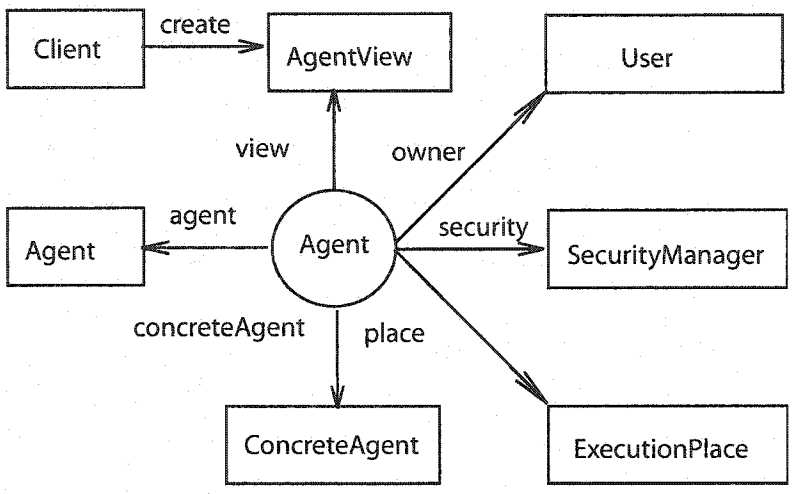

FIGURE 2.4 The Agent Pattern in AgentSpace

1. **Client**

   Manipulates agents through the AgentView reference, it can be other agents or other objects, for example Java applets.

2. **AgentView**

   The AgentView is an adaptation of the Proxy and Remote Proxy patterns. This pattern is very suitable to support transparent and secure access to the different types of objects.

   The aim of AgentView is to provide transparent access to agents. This access is done indirectly through proxies in order to protect them, and to hide transparently their current localization (this is important due to the mobility characteristic). Additionally, AgentView avoids the need to create and manage remote/virtual classes (e.g., stubs and skeletons in RMI and CORBA implementations).

3. **User**

   The user is identified by a unique identity, which contains: his/her name;

a public key; a set of certificates; the organization and country he/she belongs to; and his/her e-mail.

The agent's owner has necessarily an associated user identity, represented by an User instance.

Different users, through the corresponding AgentView instance, can access the same agent. Depending on the agent's security manager, each access is allowed or not.

4. **Agent**

   The Agent abstract class is the visible and extensible part of the Agent pattern.

5. **ConcreteAgent**

   They extend the Agent class, they implement the agent's specific functionality.

6. **ExecutionPlace**

   They define the agent's execution environment, they offer specific functions provided by the involved agent support system.

The AgentSpace system has been developed on top of the Voyager (see below) and Java Virtual Machine (JVM).

- **Aglets WorkBench**

IBM's research labs in Japan have developed an Agent based system named the **Aglets WorkBench**. The term aglet is a play on words between agents and applets. During their work, developers have recognized a number of recurrent patterns in the design of mobile agent applications. Several of these patterns were given intuitive meaning full names such as Master-Slave, Messenger, and Notifier etc, they can be conceptually divided into three

classes: traveling, task, and interaction. Danny B. Lange mentions in Lange (1998)b, that the Aglet WorkBench system "mirror the applet model in Java. The goal was to bring the flavor of mobility to the applet".

As mentioned in Lange (1998)a, the agent pattern of the Aglets WorkBench defines a set of abstractions and behavior needed to leverage mobile agent technology in Internet-like open wide-area networks. The key abstractions are Aglet, proxy, context, message, future replay, and identifier:

1. **Aglet**: An aglet is a mobile Java object that visits aglet-enabled hosts in a computer network. It is autonomous, since it runs in its own thread of execution after arriving at a host, and reactive, because of its ability to respond to incoming messages.
2. **Proxy**: A proxy is a representative of an aglet. It serves as a shield for the aglet that protects the aglet from direct access to its public methods. The proxy also provides location transparency for the aglet; that is, it can hide the aglet's real location.
3. **Context**: A context is an aglet's workplace. It is a stationary object that provides a means for maintaining and managing running aglets in a uniform execution environment where the host system is secured against malicious aglets. One node in a computer network may run multiple servers and each server may host multiple contexts. Contexts are named and can thus be located by the combination of their server's address and their name.
4. **Message**: A message is an object exchanged between aglets. It allows for synchronous as well as asynchronous message passing between aglets. Message passing can be used by aglets to collaborate and exchange information in a loosely coupled fashion.

5. **Future reply**: A future reply is used in asynchronous message-sending as a handler to receive a result later, asynchronously.
6. **Identifier**: An identifier is bound to each aglet. This identifier is globally unique and immutable throughout the lifetime of the aglet.

Behavior supported by the aglet object model is based on a careful analysis of the "life and death" of mobile agents. There are basically only two ways to bring an aglet to life: either it is instantiated from scratch (creation) or it is copied from an existing aglet (cloning). To control the population of aglets one can destroy aglets (disposal). Aglets are mobile in two different ways: active and passive. The first is characterized by an aglet pushing itself from its current host to a remote host (dispatching). A remote host pulling an aglet away from its current host (retracting) characterizes the passive type of aglet mobility. When aglets are well and running, they take up resources. To reduce their resource consumption, aglets can go to sleep temporarily, releasing their resource (deactivation), and later be brought back into running mode (activation). Finally, multiple aglets may exchange information to accomplish a given task (messaging).

The Aglets WorkBench system has been developed on top of the Java Virtual Machine.

- **Voyager**

According to Lange (1998)b and ObjectSpace (1997), ObjectSpace's **Voyager** is a platform for agent-enhanced distributed computing in Java. While Voyager provides an extensive set of object messaging capabilities it also allows object to move as agents in the network. It combines the properties of a Java-based object request broker with those of a mobile agent system. In this way Voyager allows Java programmers to create network applications using

both traditional and agent-enhanced distributed programming techniques.

2.6 Summary

The previous discussion may be summarized in the following key points:

- Using Design Patterns for framework development can largely help make a system more flexible, reusable and extensible.
- Using concurrency Design Patterns and the Mobile Agents can help resolve synchronization and network bandwidth problems, and thus significantly improve distributed system performance.
- Documenting successful design solutions stemming from concrete system implementation allows to enhance designers' vocabulary and evolve design practices.
- The space and time trade-offs are two aspects that need to be thought of when discussing the consequences for design patterns. They may also be concerned with the language and implementation issues. Since the reusability is an import factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. To study these consequences helps understand and evaluate them.

Based on these key points, the following chapters will focus on the presentation of the global architectural design of the Vador framework and document its use of patterns to implement an agent-based solution to the problem of concurrent distributed engineering for MDO.

CHAPTER 3

ARCHITECTURAL DESIGN OF THE VADOR FRAMEWORK

3.1 Global architecture of Vador

The design of a framework architecture for engineering analysis and design must allow for flexibility, scalability, and robustness. Through the layering of different services, the proposed framework architecture should allow for easy evolution of the framework as the needs evolve. Figure 1 illustrates the VADOR architecture expressed as a set of components. This architecture is divided into the classical three layers: presentation, application domain and persistent data layers. This design decouples the architecture into numerous autonomous modules in order to increase flexibility and reuse.

The VADOR modules include:

1. In the presentation layer:
    - *The VadorGUI*, which provides a graphical user interface that lets users create and manipulate interactively their own Data and Strategy Components; these components form the basis of data and process information.
    - *The DBExplorer*, which is a client-side program that provides a graphical user interface that allows communications with the *DataBase Cleint* to directly manipulate the database, where components created in the framework are stored. The *DBExplorer* is a system administration tool,

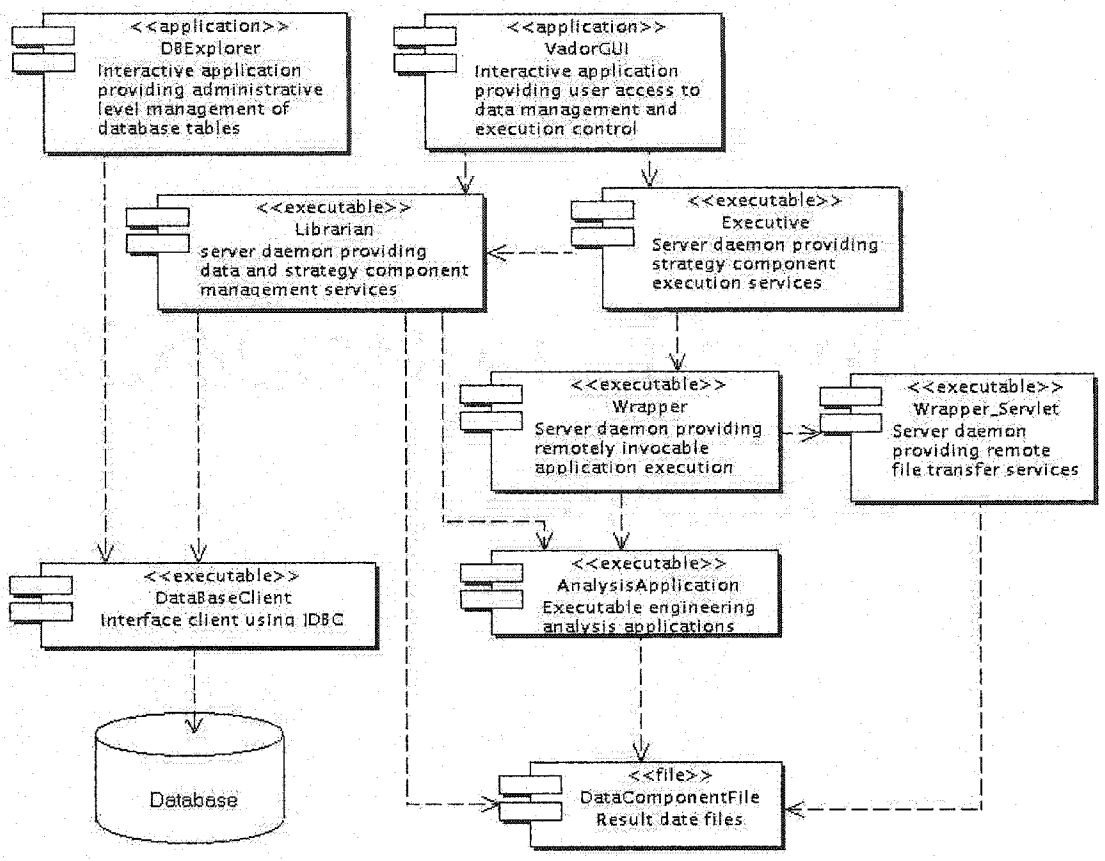

FIGURE 3.1 The global architecture of Vador

as opposed to the *VadorGUI*, which is an interface targeted toward engineering users.

2. In the application domain layer:

   - *The Librarian server*, which is responsible for the management of DataComponents and StrategyComponents, and of the interaction with the Database.
   - *The Executive server*, which is responsible for the execution of the commands issued by users through the VadorGUI, and for sending back

execution results to the Librarian when an analysis step has completed.

- *The Wrapper and the Wrapper_servlet* are the Remote CPU Servers interfaces, which are called by the Executive Server and that create the DataComponents and start the execution of the analysis applications.
- *The Analysis application* programs, which are the legacy applications that are encapsulated in the framework.

3. In the persistent data layer:

   - *The DataComponents Files*, which are the files that store results of all engineering applications,
   - *The DBMS*, which stores the descriptions of all components directly managed by the framework, including the description of DataComponents and StrategyComponents.

The next sections present the core data model of the framework, and introduce two means of encapsulation: the DataComponent, which encapsulates analysis results, and the StrategyComponent, which encapsulates analysis applications. They also briefly described the various servers, which form the basic vocabulary used to describe the VADOR framework.

3.2 The VADOR Data Model

This section presents the DataComponent and the StrategyComponent packages, which include all the base classes of the framework. The Data and the Strategy components are the basic objects which are changed and managed by the various servers and created by users in order to accomplish work through the Vador framework.

3.2.1 Strategy

The **StrategyComponent** encapsulates the engineering design and analysis methodologies in the form of executable processes. They represent the basic methods and the data flows required to transform data in a given process. The StrategyComponents are constructed using the basic elements of a procedural language including: the instantiation, test, and while constructs. The StrategyComponents can be sequential or parallel, and an IfStrategy and a WhileStrategy enable the description of complex conditional and iterative processes.

The StrategyComponent allows a user to define methods used to create DataComponents. A StrategyComponent has a type, which indicates the type of DataComponent that the StrategyComponent can create, and various attributes to indicate ownership, access, usage, history, etc. Table 3.1 contains a sample listing of Strategy attributes. Through the gathering of historical and usage information, the StrategyComponents enforce data flow standardization and process documentation.

TABLE 3.1 Strategy component attributes

Attribute	Basic content
owner	object owner
type	type of DataComponent created
usage	sample usage string of encapsulated application
history	parent(s), creation method and date
access	read and write access permissions
comments	Any comments about the object

An **Atomic Strategy** encapsulates programs which can create the data files encapsulated in the DataComponents. The programs are usually executable legacy

programs to be executed on a specific machine on the network. Note that the execution time required to run these programs can vary from a few milli-seconds to many days, depending on the specific engineering analysis to perform.

A process in the VADOR framework is a controlled sequence of programs executions which are used to produce complex data. Processes are encapsulated in **Composite Strategy** objects, which are defined by users using standard elements of structured procedural programming languages: sequential blocks, parallel blocks, if constructs and controlled loops.

3.2.2 DataComponent

DataComponents are objects that encapsulate engineering design and analysis data which are contained in actual data files. They comprise set of attributes required for data management, but leave the data itself in the files that are being encapsulated. A DataComponent can be atomic (i.e. encapsulate only one data file) or composite (i.e. encapsulate other composite or atomic DataComponents). A DataComponent requires a StrategyComponent to fill its encapsulated data file.

Two related classes form the basis of the DataComponents representation. The first class is the **DataComponent Definition** which defines the behavior of the DataComponent, such as which type of file can be encapsulated, which execution order can be used, etc. The DataComponent definition is an abstract class that is never directly instantiated. We named it as "DCType". The other class is the **DataComponent Instance**, which governs a concrete object that encapsulates concrete files. We named it as "DCInstance".

Figure 3.2 presents the classes in the DataComponent package.

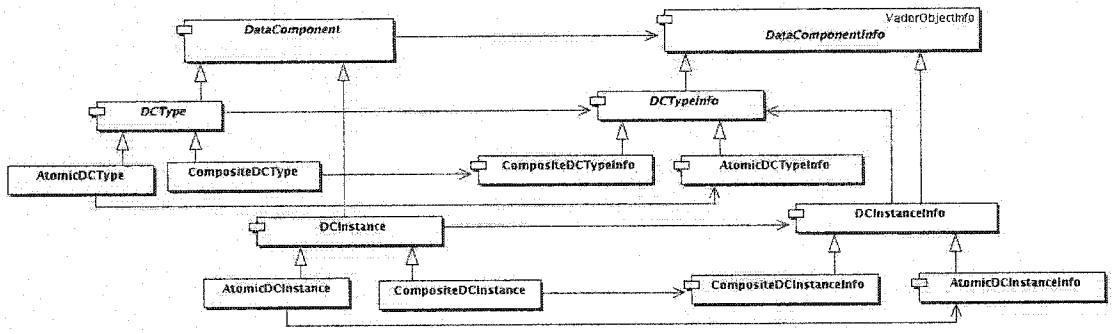

FIGURE 3.2 The DataComponent package UML diagram

DCType is the DataComponent definition class. We define two kinds of DCType: AtomicDCType and CompositeDCType. The DCTypeInfo class keeps the DCType attributes, the AtomicDCTypeInfo keeps the AtomicDCType attributes and the CompositeDCTypeInfo keeps the CompositeDCType attributes.

DCInstance is the DataComponent instance class. There are also two kinds of DCInstance: AtomicDCInstance and CompositeDCInstance. The DCInstanceInfo class keeps the DCInstance attributes, the AtomicDCInstanceInfo keeps the AtomicDCInstance attributes and the CompositeDCInstanceInfo keeps the CompositeDCInstance attributes. The DCInstanceInfo also has a DCTypeInfo object in its attribute to keep the DCType's attribute value in its object.

All the DCType and DCInstance attribute values are stored in the Vador Database. A detailed database UML diagram can be found in appendix I.

There is a Composite Design Pattern in the DataComponent structure. It will be described in the **Use of the Composite Pattern** section, in chapter 5.

Table 3.2 presents a sample listing of member attributes for the DCInstance class. Through the centralized management of result file history and ownership, the

framework effectively implements traceability of outputs produced under its control. This represents one of the most fundamental attributes of a multidisciplinary environment.

TABLE 3.2 Data component instance attributes

Attribute	Basic content
owner	object owner
strategy	Identifier of the creation strategy
url	location of encapsulated data file
history	parent(s), creation method and date
access	access permission
status	data and methods status
comments	Any comments about the object

3.3 Librarian Server

The **Librarian Server** allows to decouple the VADOR API from the underlying Database server capabilities.

The Librarian is a Java program with capabilities for handling Components by providing an interface to a suitable Database Management System in order to store the pertinent information about each Component. It is important to emphasize that the present architectural design supports the separation of the basic data, usually contained in files and potentially rather large, from the descriptive information. Typically, only the descriptive information, or Meta-data, will be stored in the Database. The large datasets will be stored on separate file servers and will usually reside where they have been created.

3.4 Executive Server

The **Executive Server** is a Java program that manages the execution of StrategyComponents, which result in the creation of DataComponents.

The Executive server interacts with the Librarian in order to retrieve DataComponents to be created. It then triggers and monitors Component creation through the remote execution of legacy applications, and finally updates the database contents.

Executive servers can be run in any number on the network, and can run on any machine. A user can therefore connect to the closest Executive server running on the network and send execution commands that will then be fully taken in charge by the Executive server. This allows the user to shutdown its connection with the framework, while his submitted jobs remain active.

3.5 Wrapper Server

The **Wrapper Servers** are the Remote CPU Servers interfaces, which are called by the Executive Server in order to start the execution of the analysis applications and create the DataComponents files. A complete description of the Wrapper Server can be found in Zhou (2003).

3.6 VadorGUI

Most users will interact with the VADOR framework through its GUI.

The **Vador GUI** is a Java program which will be running on the user's machine. Because of Java's portability, all users will be running the same program, inde-

pendently of their particular machines. However, the appearance and feel can be different, based on Java's own particularities.

The next two chapters will discuss the pattern based architecture of the Vador framework. Most patterns used in the Vador system are described in the books by Gamma *et al.* (1994) and Schmidt *et al.* (1999).

CHAPTER 4

ARCHITECTURE OF THE VADORGUI MODEL

This chapter aims to explain in details the functionality of the VadorGUI model and discuss design decisions at the root of the model.

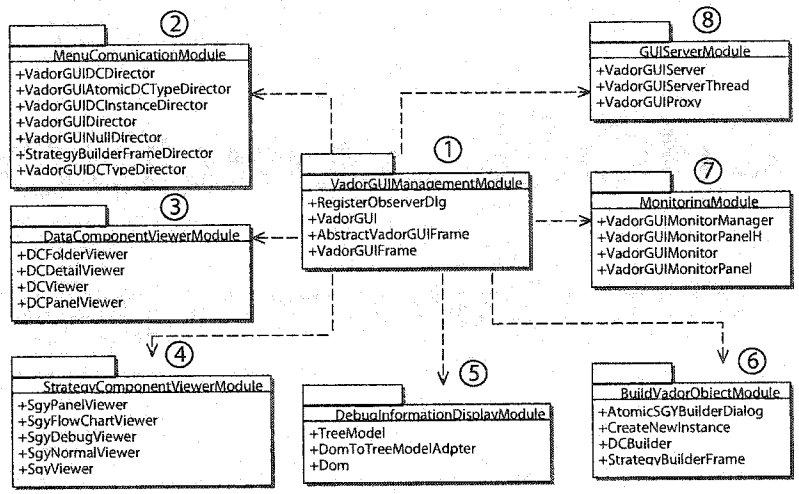

FIGURE 4.1 The UML of the VadorGUI Model

The objective of this chapter is to document key extension points in the VADORGUI architecture and illustrate the use of a number of important patterns, namely, Mediator, Template Method and Adapter. Figural 4.1 shows a high level diagram of the packages that form the VadorGUI Model. These packages form the presentation layer of the system.

The individual packages will be discussed in the following subsections:

1. The VadorGUI Management module

2. The Menu Communication module
3. The DataComponent Viewer module
4. The StrategyComponent Viewer module
5. The Debug Information Display module
6. The VadorObjects Builder module
7. The Monitoring module
8. The GUI Server module

4.1 The VadorGUI Management module

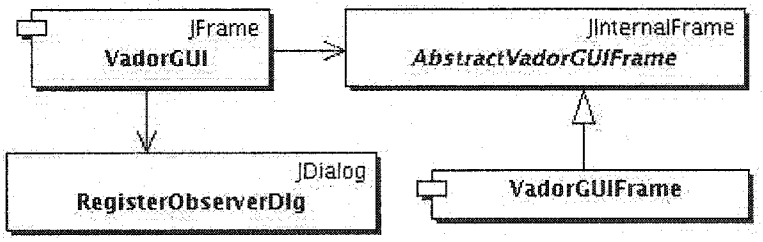

FIGURE 4.2 The VadorGUI Management Module classes diagram

Figural 4.2 presents the classes diagram in the VadorGUI Management Module package.

The main classes in this package are the VadorGUI and the VadorGUIFrame classes. The main task of the VadorGUI class is to start and manage the VadorGUI application and its Server, display the main VadorGUI window, and keep the objects

of the VadorGUIFrame class. The VadorGUIFrame class and the StrategyBuilder-Frame class (in the Builder classes Module) are the sub-classes of the Abstract-VadorGUIFrame class which defines a sub-window of the main application window. The VadorGUIFrame class displays Vador objects such as DCType, DCInstance and Strategy objects in special sub-windows. The RegisterObserverDlg class displays a model dialog to allow users to register themselves to interesting events that they want to be notified about.

4.2 The Menu Communication module

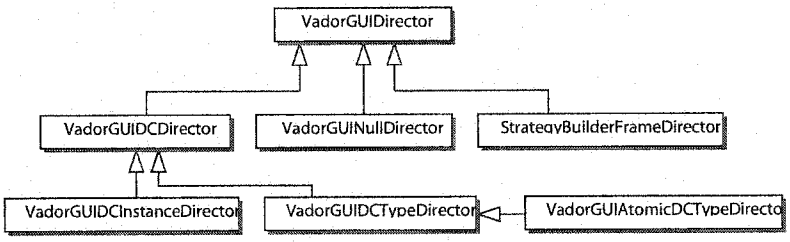

FIGURE 4.3 The Menu Communication Module classes diagram

Figural 4.3 shows the Menu Communication Module classes. These classes define communication mechanisms that allow interactions between the different menu objects; they work together as a **Mediator** pattern.

Mediator Pattern Description The menus must be configured according to the type of object displayed in the GUI frame. When there are no objects in the GUI frame, most menus should be disabled, while when an object is being opened, changes to the menu's state depends on the type of object that is going to be displayed. There are many interconnections between the menu objects and the type of objects displayed.

This collective behavior has been encapsulated in a separate mediator object.

The **Mediator pattern** defines an object that encapsulates how a set of objects interact. The mediator promotes loose coupling by refraining objects from referring to each other explicitly, and by letting their interaction vary independently.

Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; and, in the worst case, every object ends up knowing about every other.

Although partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again. Lots of interconnections make it less likely that an object can work without the support of others, the system then acts as though it were monolithic. Moreover, it can be difficult to change the system's behavior in any significant way, since behavior is distributed among many objects. As a result, numerous interconnections may force definition of many subclasses in order to customize the system's behavior.

A mediator is responsible for controlling and coordinating the interactions of a group of objects. The objects only know the mediator, thereby reducing the number of interconnections.

In our framework, the VadorGUIDCTypeDirector is the mediator between the various menu objects according with the DCType objects that are displayed in the VadorGUI frame. A VadorGUIDCTypeDirector object knows the menu objects and coordinates their interaction. It acts as a hub of communication for menus.

For example, when the user clicks on the 'OpenDC' item in the menu (OpenDC-Menu in figure 4.4) and opens a DCType object in the VadorGUIFrame window, the DCDetailViewMenu object should be activated to let the user select it. But

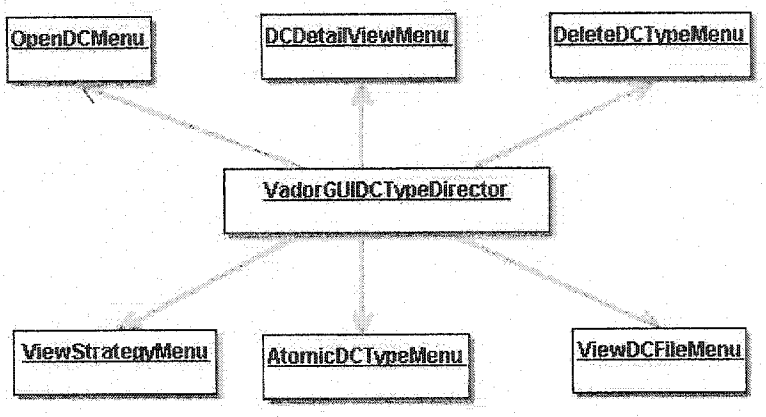

FIGURE 4.4

the OpenDCMenu and DCDetailViewMenu objects don't know each other. The OpenDCMenu object thus sends a notification to the VadorGUIDCTypeDirector, and it is the VadorGUIDCTypeDirector who will notify the DCDetailViewMenu object to activate itself.

The director mediates between the OpenDCMenu and the other menus, which allows the menus to communicate with each other only indirectly, through the director. They don't have to know about each other; all they know is the director. Furthermore, because the behavior is localized in one class, it can be changed or replaced by extending or replacing that class.

The VadorGUIDirector is an abstract class that defines the overall behavior of the change of the menu's states. The operation setVadorGUIMenuEnabled is an abstract operation for changing the menu's states, it will be called when a Data-Component is opened or closed. The VadorGUIDirector subclasses override this method to create the proper behavior of the menu state changes.

The concrete VadorGUINullDirector class implements the behavior for menu states

changes when there is no object displayed in the GUI frame, and the concrete-VadorGUIDCTypeDirector class implements the same behavior when a DCType object is opened in the GUI frame.

Participants

- The VadorGUIDirector acts as a **Mediator** and defines an interface for communicating with the Menus objects. This is an abstract class.
- The VadorGUIDCTypeDirector, VadorGUIDCInstanceDirector, VadorGUINullDirector are **ConcreteMediator(s)** that implement cooperative behavior by coordinating Colleague objects, they know and maintain colleagues relationships between the menu objects.
- The menu objects are the concrete **Colleague classes**. They know the Mediator object. Each menu object communicates with its mediator whenever it would have otherwise communicated with another menus.

Collaborations The menus send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate menus.

Consequences This mediator localizes behavior that otherwise would be distributed among several objects. Changing the interaction behavior between the menus requires sub-classing the Mediator only; The Menu classes can be reused as they are. This effectively decouples the Menus and simplifies object protocols among them.

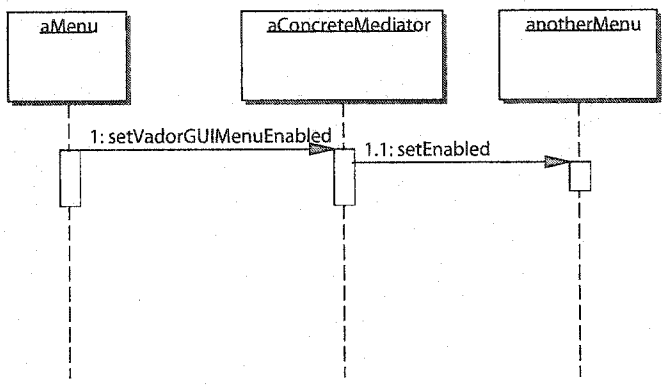

FIGURE 4.5 The Mediator Pattern sequence diagram

4.3 The DataComponent Viewer module

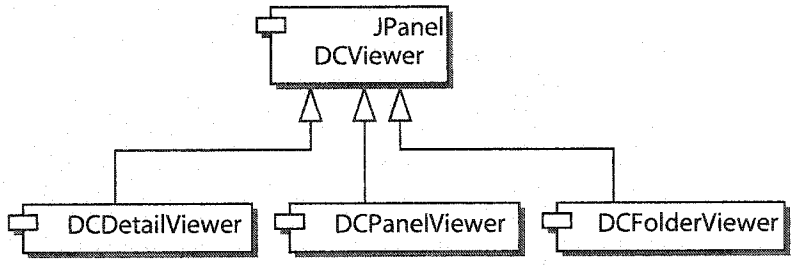

FIGURE 4.6 The DataComponent Viewer Module diagram

Figure 4.6 shows the classes in the DataComponent Viewer Module package.

The DataComponent viewer module defines the classes and functions to display the DataComponent objects under different graphical representations. They work together using the **Template Method** pattern.

Template Method Pattern Description The VadorGUI application provides a graphical user interface that lets users create and manipulate DataComponents and StrategyComponents. One of the main objective of the whole GUI is to present specialized views of the various objects that the user can manipulate in the framework. To this end, the Viewer Model provides several classes that can each reconfigure the object viewing area according to user preferences.

For example, the DCDetailViewer, the DCFolderViewer and the DCPanelViewer allow to display DataComponent objects with varying levels of details and information content. The DCDetailViewer displays the detailed attributes of the DataComponents in a table. The DCFolderViewer displays the DataComponents as small icons, and the DCPanelViewer displays the DataComponents as big icons. To create and manage these viewer, a set of uniform steps are used, but the behavior associated to each step is different among the different types of viewers. Most viewing algorithms are implemented as Template Methods according to the pattern that bears that name.

The **Template Method Pattern** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Methods let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Abstract DCViewer class defines the general steps (algorithm) of the viewer creation and manipulation with each step implemented as a call to a special abstract function. The concrete subclasses redefine these functions to suit specific needs. For example, the DCDetailViewer class redefines the functions to implement a list-based view of the DataComponent with configurable list of details, and the DCFolderViewer class implements an icon-based view of the DataComponent.

The abstract DCViewer class defines the algorithm for creating a viewer in its

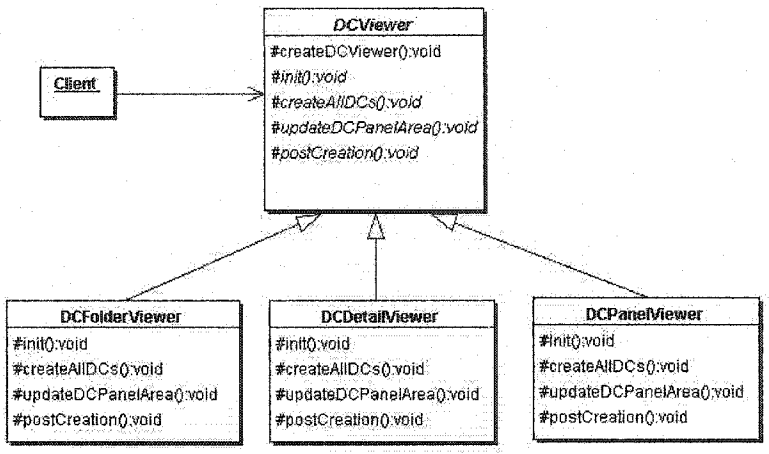

FIGURE 4.7

createDCViewer operation:

```
protected void createDCViewer() {
    init();
    createAllDCs();
    DataComponent dc = parent.getSelectedDC();
    updateDCPanelArea(dc);
    postCreation();
}
```

The function createDCViewer defines the steps for creating a DCViewer. It initializes the attributes, creates the special swing objects for the concrete viewer, gets the DataComponent, shows the concrete viewer, etc.

The createDCViewer operation is a template method. It defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. DCViewer subclasses define the steps of the algorithm that initialize the attributes,

and creates the specialized views of the DataComponent. The concreteDCViewer template method also defines an operation – postCreation() where subclasses can define further behavior once their viewer creation has completed.

By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering, but lets the subclasses vary those steps to suit their needs.

Participants

- DCViewer is the **AbstractClass** that defines the abstract primitive operations (init(), createAllDCs(), updateDCPanelArea(), postCreation()) that concrete subclasses define to implement steps of the algorithms. The DCViewer class also implements a template method (createDCViewer()) defining the skeleton of the creation algorithm. The template method calls primitive operations as well as operations defined directly in the DCViewer class and in other objects.
- DCDetailViewer, DCFolderViewer, DCPanelViewer are the **ConcreteClasses** that implement the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations The ConcreteClasses rely on the AbstractClass to implement the invariant steps of the algorithm.

Consequences Template methods are a fundamental technique for code reuse. Template methods lead to an inverted control structure where a parent class calls

the operations of a subclass and not the other way around. This type of inverted control makes the design simpler to understand and more robust.

4.4 The StrategyComponent Viewer module

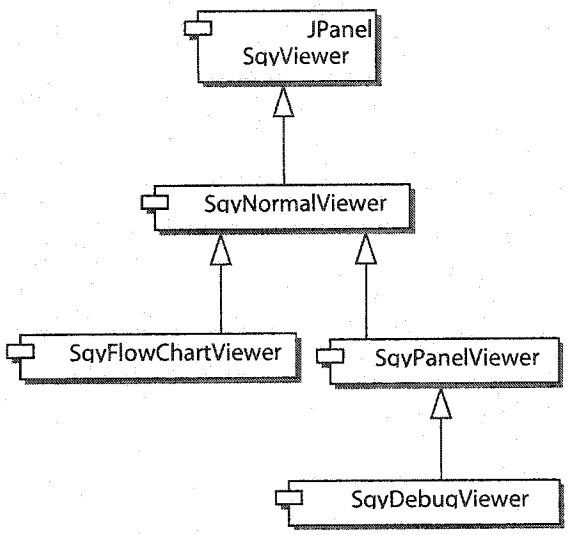

FIGURE 4.8 The StrategyComponent Viewer Module diagram

Figure 4.8 illustrates the structure of the StrategyComponent Viewer module.

This set of classes implement the StrategyComponent viewer functionality. The architectural design of this module is closely related to the one used for DCViewer Module, and uses the similar architecture.

4.5 The Debug Information Display module

Figure 4.9 shows the classes in the Debug Information Display Module package.

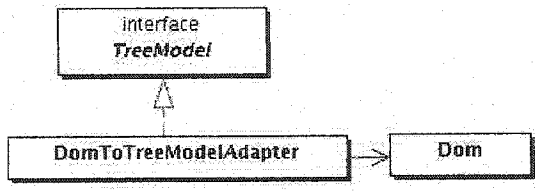

FIGURE 4.9 The Debug Information Display Module diagram

These classes define the functionality to transfer the XML DOM node to the Java JTree node (see the Description item below). Their design is based on the **Adapter** pattern.

Adapter Pattern Description When a StrategyComponent is executed on the Executive Server, errors may occur, for a variety of reasons. Sometimes, the error comes from the user side (the user may make mistakes in his scripts), sometimes, they are caused by the system (the input files to be used may not exist). The Vador Framework provides a debugging tool to assist users in locating and fixing the bugs in their strategies. This debugging tool keeps track of all the execution information generated by the framework, and stores this information in an XML file. When the user needs to diagnose an error, the debugging tool displays the execution information (including both normal information and errors) in the VadorGUI. As XML data can be represented in a DOM node, which is a tree, the Java swing – JTree class can be used to display it. But since the DOM node interface is different from the JTree's TreeModel interface, the Adapter pattern must be used to convert the DOM node interface into the TreeModel interface.

The **Adapter pattern** aims to convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

The DOM node interface is different from the JTree's TreeModel. The issue then becomes how to make an existing and unrelated class like the DOM node work in the VadorGUI that expects classes with a different and incompatible interface. Changing the DOM node class so that it conforms to the TreeModel interface is of course impractical, since it would require modifying the toolkit's source code. Even if the code were available, it would still not make sense to change the DOM node class; the toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.

Instead, a class named DomToTreeModelAdapter has been inserted in the design that adapts the DOM node interface to the TreeModel interface. This can be done in one of two ways: (1) by inheriting TreeModel's interface and DOM node's implementation or (2) by composing a DOM node instance within a DomToTreeModelAdapter and implementing the DomToTreeModelAdapter in terms of the DOM node's interface. These two approaches correspond to the class and object versions of the Adapter pattern, and in both cases the DomToTreeModelAdapter is called an adapter.

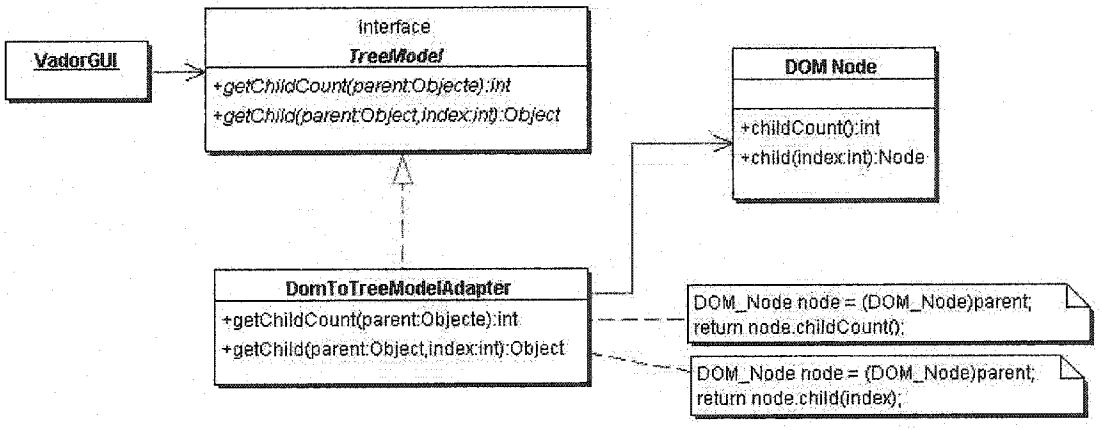

FIGURE 4.10 The adapter pattern class diagram

Figure 4.10 illustrates the class diagram for the object adapter case. In the VadorGUI, this second method has been chosen because it lets a single Adapter work with many Adaptees, that is, the Adaptee itself and all of its subclasses, if any. The Adapter can also add functionality to all Adaptees at once.

The object adapter relies on object composition. The TreeModel interface is a standard interface used in the VADOR Viewers, since the DomToTreeModelAdapter class adapts the DOM node class to the TreeModel interface, the VadorGUI can reuse the otherwise incompatible DOM node class in the VADOR Viewers.

Participants

- TreeModel is a **Target** that defines the domain-specific interface that Client uses.
- VadorGUI is a **Client** that collaborates with objects conforming to the Target interface.
- DOM node is an **Adaptee** that defines an existing interface that needs adapting.
- DomToTreeModelAdapter is an **Adapter** that adapts the interface of Adaptee to the Target interface.

Collaborations The VadorGUI calls operations on the DomToTreeModelAdapter instance. In turn, the DomToTreeModelAdapter calls DOM node operations that carry out the request.

Consequences The adapter lets the classes that have the incompatible interfaces work together. Although, this use of the adapter pattern implies an extensibility trade off. Indeed the adapter makes it harder to override Adaptee behavior. It will require sub-classing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

4.6 The VadorObjects Builder module

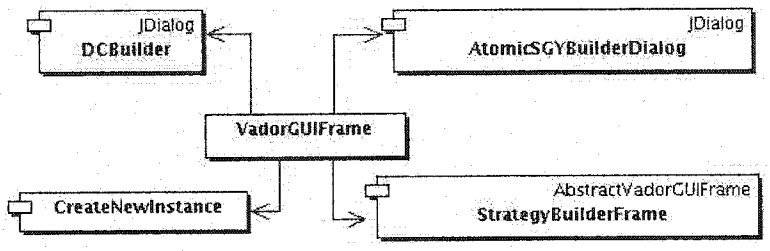

FIGURE 4.11 The VadorObjects Builder Module diagram

Figure 4.11 shows the classes in the VadorObjects Builder Module package.

These are the builder classes, they define the functionality to build the Vador objects in the VadorGUI. Vador objects include DCType, Strategy and DCInstance objects.

1. The DCBuilder lets the users create DCType object (atomic or composite).
2. The AtomicSGYBuilderDialog class defines a dialog that allows users to create Atomic Strategy objects.
3. The StrategyBuilderFrame class offers the tools to help users create the different types of Composite Strategy objects, such as Sequential Strategy object, Parallel Strategy object and so on.

4. The CreateNewInstance class allows users to create the DCInstance objects.

4.7 The Monitoring module

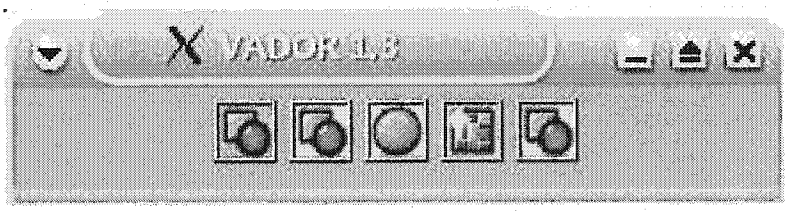

FIGURE 4.12 The VadorGUI Monitoring window

Figure 4.12 shows the VadorGUI Monitoring window. It is a small separate window which provides monitoring functionalities, such as management of displaying status of the sub-windows in the main VadorGUI window and showing the new message arrival signals.

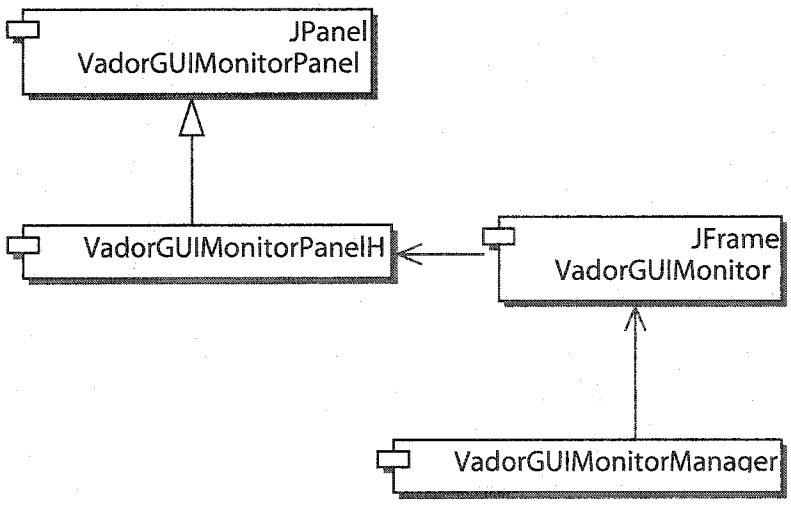

FIGURE 4.13 The Monitoring Module diagram

Figure 4.13 shows the classes in the Monitoring Module package. These classes create and manage the VadorGUI Monitoring window.

4.8 The GUI Server module

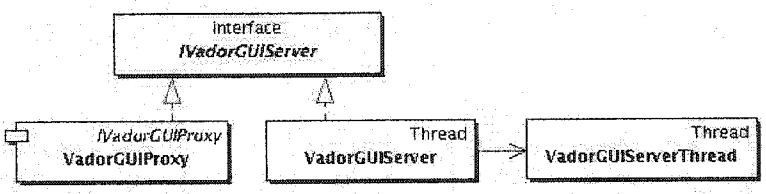

FIGURE 4.14 The GUI Server Module diagram

Figure 4.14 shows the classes in the GUI Server Module package.

These are the VadorGUI server classes. They work together to offer the server function on the VadorGUI side, they listen to a port number, receive and deal with the requests from the other servers. Details on the Server architecture are presented in the next chapter.

CHAPTER 5

GLOBAL APPLICATION LAYER ARCHITECTURE – THE AGENT PATTERN

The application domain layer of the framework comprises two main servers, the Librarian Server and the Executive Server. This layer constitutes the core of the Vador System, this is the layer where resides most of the functionality to process data and tasks within the framework. This chapter presents the architecture of this layer, which is implemented using a specialization of the **Active Agent** pattern.

5.1 Pattern description

The **Active Agent pattern** decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control. It also decouples the method execution from the execution platform: Method executions are thus encapsulated in mobile agents. This pattern is based on the Active object, Command, Proxy, Visitor and Strategy patterns.

The Active Agent pattern is used to solve problems related to the concurrency, scability and flexibility of the framework.

5.2 Applicability

Use of the Active Agent pattern in the context of VADOR is justified by the following:

- VADOR needs to define autonomous active-objects supported by the framework.
- VADOR needs to allow clients (such as the VADORGUI) to ignore low-level details, such as distribution or security aspects, for instance to obtain references to agents and to interact with agents transparently.
- VADOR needs to simplify as much as possible the development and management of dynamic and distributed applications in a consistent and documented way.
- VADOR needs to solve concurrency, scability and flexibility problems.

5.3 Architectural issues

The active Agent pattern implements the agent support system itself, it doesn't use any other agent system. This choice has several architectural implications:

1. The chosen Agent programming language is Java.
2. The chosen Agent communication language between the Agents and Servers is based on Java messages.
3. Security and access control: in an open and distributed environment, these

issues are very important and are currently being investigated as an autonomous topic.

4. The chosen low level distribution support mechanism is implemented directly in the Vador system.

5. VADOR implements mobility of the agents. Agents can navigate through the different Vador Servers in order to communicate locally with the server and execute the tasks. The Vador Framework implements a constrained mobility model, where after migration, the agent must be restarted.

6. Communication between the Agents: as the Vador System is a relatively simple environment, this type of communication is currently not necessary nor available (we only have the communication between the Agents and Servers).

5.4 The agent based system: Vador System

The whole Vador system is a specialized **Active Agent Pattern**. It is implemented on top of the Java Virtual Machine. Figure 5.1 shows the main components in the Vador System. Both server and client run on top of Java Virtual Machine(JVM), they can run in the same or different machines. Agents run on the Vador Servers, they interact with their end-user through the VadorGUI.

5.5 Structure and participants

Figure 5.2 shows the Active Agent pattern collaboration diagram and identifies internal participants to the pattern, that cooperate to provide services to an external participant, the client. Following is a brief description of each participant,

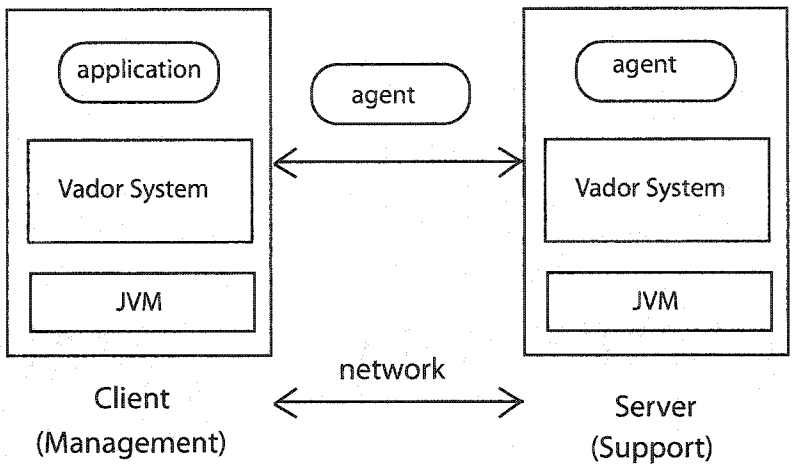

FIGURE 5.1 The Active Agent Pattern components

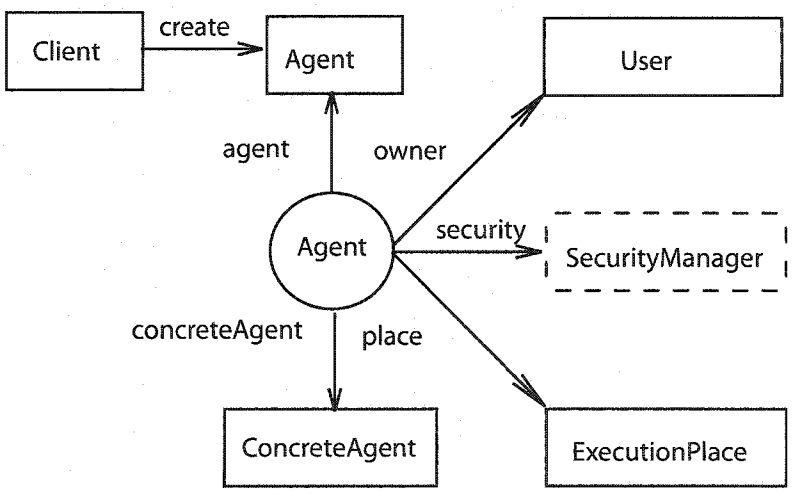

FIGURE 5.2 Structure of the Active Agent pattern

presented in relation with actual instances of the pattern in the Vador framework.

- **Client**

  The client creates and manipulates the agents using the standard interfaces provided by the pattern. In the context of the VADOR framework, Clients

represent applications that directly perform tasks on behalf of the framework users, such as the VadorGUI model.

- **User**

  The User is identified through a unique identifier (id) in the system. When the user creates an agent through one of the VADOR applications, his id is included in the agent, and this agent becomes his delegate.

- **Agent**

  The Agent abstract class is the visible and extensible part of the Active Agent pattern, it defines the abstract behavior of the agent, which includes, in the case of the VADOR framework, the **call** function (concrete agents begin their executions on invocation of this function) and the **can_run** function (concrete agents define their execution behavior by overloading this function).

- **ConcreteAgent**

  Concrete agent classes are Agent subclasses. They implement the behavior related functions to execute the concrete tasks. For instance, the OpenStrategy Agent loads the StrategyComponent object from the Database, the SaveStrategy Agent saves the StrategyComponent object in the Database.

- **SecurityManager**

  This class specifies the Agent access security policy, it controls all the operations made available on the agent component. This class is planned but not currently implemented.

- **ExecutionPlace**

  This class specifies the agent's computational environment, which corresponds to the place where it was created as well as where it is currently

resident. In the Vador framework, the Execution Places are the Vador Servers (the ExecutiveServer, the LibrarianServer).

The following sections will discuss the Active Agent Pattern components.

The Active Agent Pattern has three main components: The Vador Server, the Client and the Agent. The Client component (VadorGUI) has already been described in the previous chapter, the Agent component will be described next, followed by a discussion on its implementation. Then the Server component will be described, and the proposed overall architecture will be compared to related work.

5.6 Architecture of the Agent component

The **Agent** component is structured as a package which includes the Java interfaces and classes. It is an active object belonging to a user and capable of executing on some Vador Server. The Agent has all the data and execution methods needed to complete its tasks on the Vador Server and it can communicate with other Vador Servers in local or remote places. After its creation, it begins to navigates from one Server to another, collecting the data and executing the tasks for its client.

5.6.1 The sub-components in the Agent component

As Figure 5.3 shows, the Agent component comprises three sub-components. The Vador Object component carries the operation data, the Vador Visitor components include the information on how to operate on the data and the Vador Itinerary components contain the mobility information.

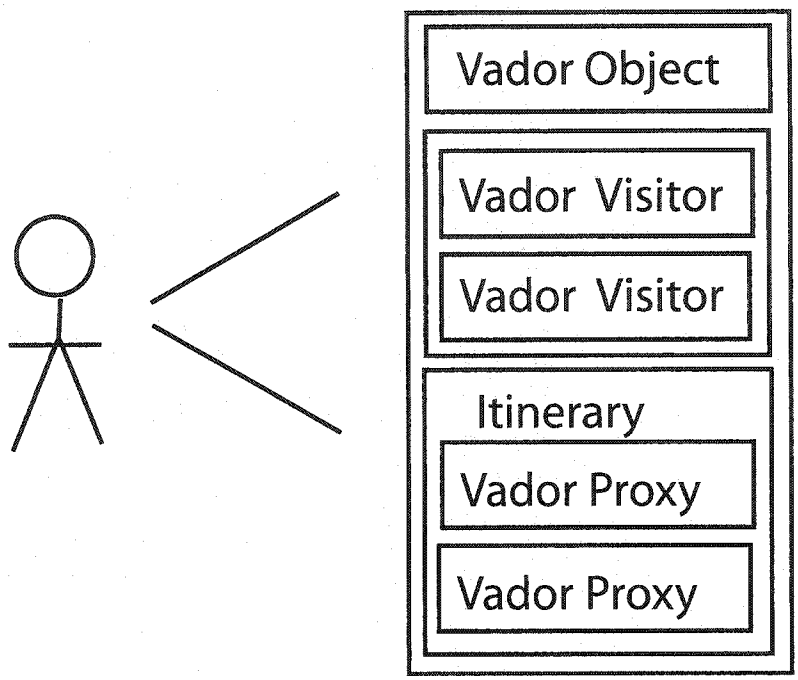

FIGURE 5.3 The components in the Agent component

5.6.2 Vador Object

The Vador Object carries the operation data. This is an abstract component which carries the actual data on which the agent needs to operate. The Vador framework defines several different types of Vador Object, StrategyComponent, DataComponent, VadorComment, VadorHistory and VadorObserver. The StrategyComponent encapsulates the design-and-analysis methodologies or processes. The DataComponent encapsulates design-and-analysis data in actual data files, it has two sub classes: DCTypeComponent and DCInstanceComponent. The VadorComment encapsulates the comments about different types of Vador components: the DCTypeComponent has the DCTypeComment, the DCInstanceComponent has the DCInstComment and the StrategyComponent has the StrategyComment. They are implemented using the **Composite** and the **Strategy** pattern.

5.6.3 Vador Visitor

The Vador Visitor includes the information on how to execute a task.

It is an execution class. We use the **Visitor** pattern to implement this component. The Visitor represents an operation to be performed on the elements of an object structure. It allows to define a new operation without changing the classes of the elements on which it operates. Different execution behaviors can be defined on Vador objects, such as Vador Saving Visitor, Vador Opening Visitor and Vador Deleting Visitor. Each corresponds to one operation that can be remotely defined and executed.

Vador can have one larger task which includes several small tasks. In this case the Vador Composition Visitor is used. The Vador Composition Visitor is an object which includes several other Vador Visitor objects in a fixed order. This enables several tasks to be executed (at the same or different locations) in a pre-defined order.

5.6.4 Vador Itinerary

The Vador Itinerary contains the mobility information.

It includes several Vador Proxy objects. Each Vador Proxy represents a destination server that the Agent wants to go to. The Itinerary object encapsulates the agents' itineraries and their navigation among multiple destinations.

Being an autonomous mobile entity, an agent is capable of navigating by itself independently to multiple hosts. Consequently, it is probably preferable to separate the handling of navigation from the agent's behavior and message handling, thus

promoting modularity of every part. The Itinerary component was used to shift the responsibility for navigation from the agent object to an Itinerary object. The itinerary class provides an interface to maintain and modify the agent's itinerary and to dispatch it to new destinations. An agent object and an Itinerary object are connected as follows: the agent creates the Itinerary object and initializes it with (1) a list of destinations to be visited sequentially and (2) a reference to the agent; then, the agent uses the "go" method to dispatch itself to the next available destination in its itinerary or back to its origin, respectively. To support the above, it is necessary that the Itinerary object be transferred together with the agent, and that their references to each other be maintained at every destination.

The Vador Proxy object is implemented using the **Proxy** pattern.

5.7 Implementation of the Agent component

The Agent component package diagram is shown in the Figure 5.4. The individual packages will be discussed in the following subsections:

1. The Abstract Agent module
2. The Vador Object module
3. The Visitor module
4. The Proxy module
5. The Concrete Agent module
6. The Agent Tools module

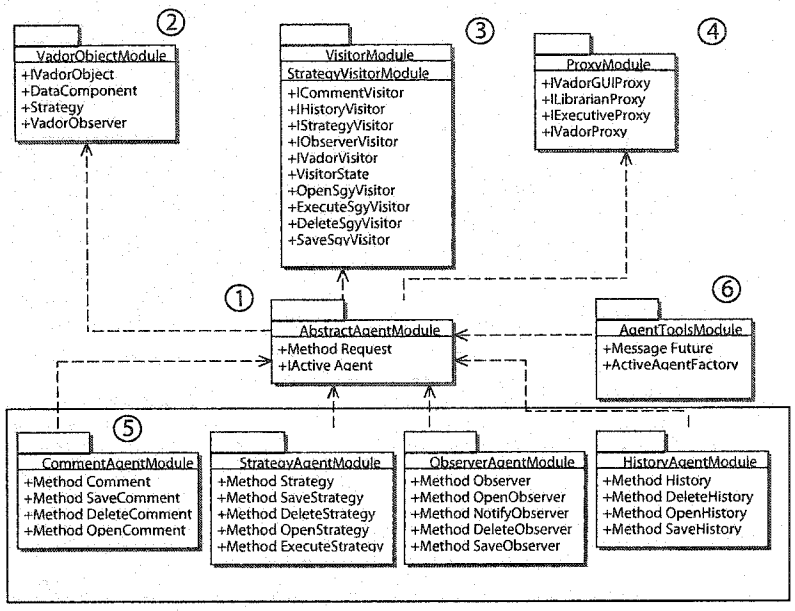

FIGURE 5.4 The Agent component package diagram

5.7.1 The Abstract Agent module

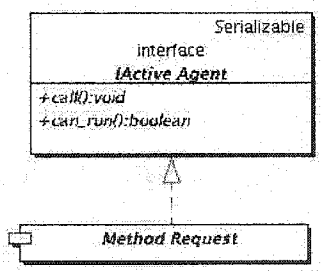

FIGURE 5.5 The Abstract Agent Module classes diagram

Figure 5.5 presents the class diagram of the Abstract Agent Module package.

This package defines the IActive_Agent interface and the Method_Request class. They define the agent skeleton.

5.7.1.1 IActive_Agent interface

Class IActive_Agent	Collaborator
Responsibility • Represents a method call on the active object • Provides guards to check whether the method request becomes runnable	

FIGURE 5.6 IActive_Agent

This interface defines the interface of the Active Agent pattern, it has two abstract operations, they are the "call" operation and the "can_run" operation. The "call" operation invokes the execution of the Vador object who is encapsulated in the Active Agent. The can_run operation is a guard method that can be used to determine if an Active Agent can be executed. This interface is similar to the **Command** pattern.

Command Pattern Description The Vador project is multi-tiered client-server architecture framework. It accomplishes work by sending the Agent object through the network to change and manage the Data. When the agent arrives at the Server, it is not necessary for the Server to know anything about how to act on the Agent object.

The Command pattern lets senders and receivers exchange requests of unspecified

type by turning the request itself into an object. This object can be stored and passed around like other objects.

The **Command Pattern** encapsulate a request as an object, thereby allowing clients to be parameterized with different requests, queue or log requests, and support undoable operations.

The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form, this interface includes an abstract Execute operation. Concrete Command subclasses specify an action by implementing the Execute operation. The receiver has no knowledge required to carry out the request, it just calls this Execute operation to invoke its execution.

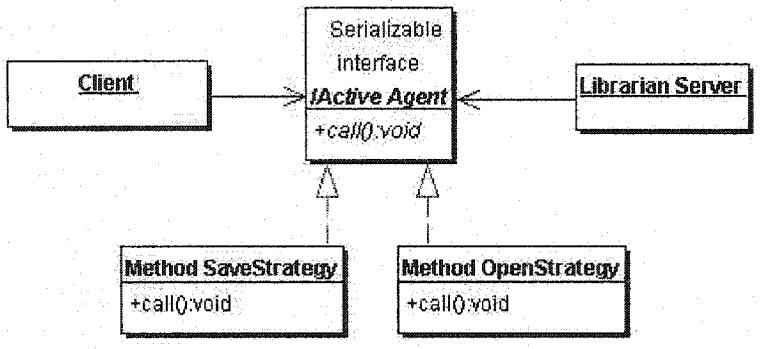

FIGURE 5.7

The class IActive_Agent is the abstract Command class, it declares a call function which indeed is an Execute operation, the concrete classes such as Method_SaveStrategy implement this function to carry out a concrete job (save the StrategyComponent in the database). For example, the client creates the Method_SaveStrategy object by inserting a StrategyComponent object in it, then sends it to the Librarian Server, the Librarian Server receives this object and calls its Execute operation, thereby allowing the Method_SaveStrategy object to save

the contained StategyComponent in the database.

Notice how the Command pattern decouples the object that execute the operation from the receiver. New commands can be added without modifying the receiver as the receiver has no knowledge about the concrete command.

Participants

- **Command** (IActive_Agent) declares an interface for executing an operation.
- **ConcreteCommand** (Method_SaveStrategy, Method_OpenStrategy) implements a concrete action.
- **Client** (Application) creates a ConcreteCommand object.
- **Receiver** (VADOR Server) invokes the operation without knowing how to perform the operations associated with carrying out a request. Any Vador framework Server may act as a Receiver.

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- A receiver object receives the ConcreteCommand object.
- The receiver invokes operations by calling Execution operation on the command.
- The ConcreteCommand object executes the operations on its receiver to carry out the request.

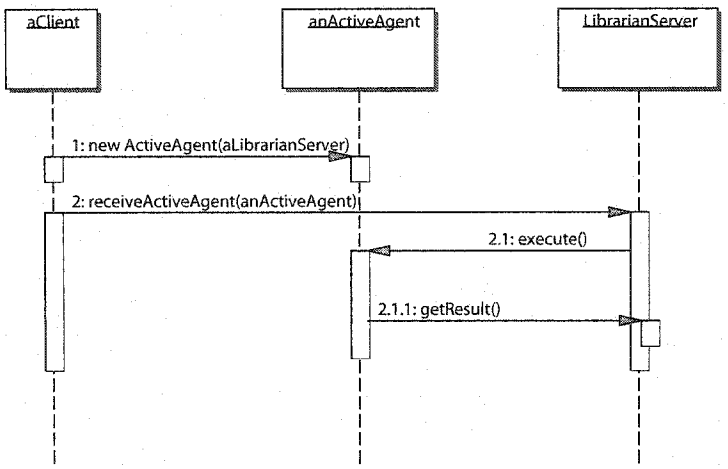

FIGURE 5.8 The Command Pattern sequence diagram

Consequences The Command pattern has the following consequences:

- Command decouples the object that invokes the operation from the one that knows how to perform it. It decouples the concrete VadorObject (see next section) object and the VadorServer (see section after) object in the Vador framework.
- Commands are true objects. They can be manipulated and extended like any other object. There are different concreteAgents extending from the Abstract Agent which can be differently manipulated in the Vador framework.
- The commands can be assembled into a composite command. With this characteristics we can have a special concrete agent which includes several other concrete agents if necessary.
- It is easy to add new commands, because we don't have to change existing classes. As a result, we can design any new concrete agent without affecting

the existing agents.

5.7.1.2 Method_Request class

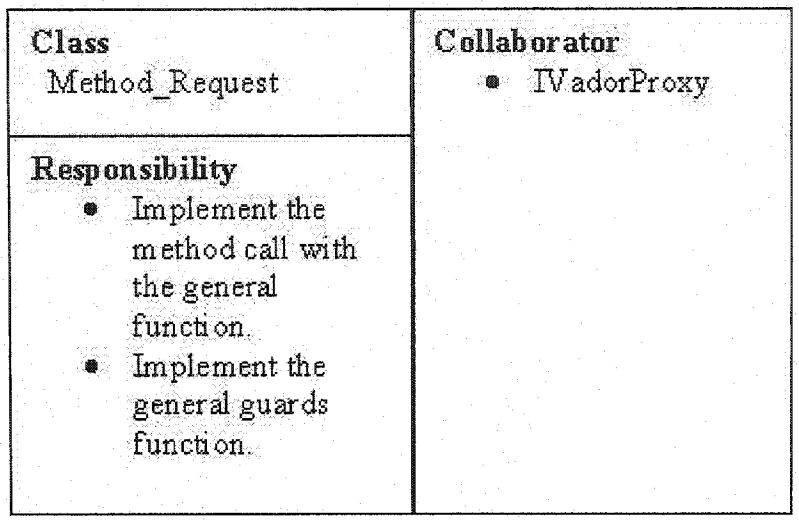

FIGURE 5.9 Method_Request

This abstract class implements the Vador_Agent interface. It defines general execution rules for the Vador Object. A Method_Request contains several components:

1. the context information, such as the parameters necessary for executing a specific method. They are the Vador Object components.
2. the execution interface–Vador Visitor interface (IVadorVisitor).
3. the itinerary interface–Vador Proxy interface (ILibrarianProxy, IVadorGUIProxy).

The Method_Request must be light, as it works as a mobile agent. So it just

contains the interfaces of the Vador Visitor and Vador Proxy, the concrete classes will be dynamically loaded when the agent arrives on the Vador Server.

5.7.2 The Vador Object module

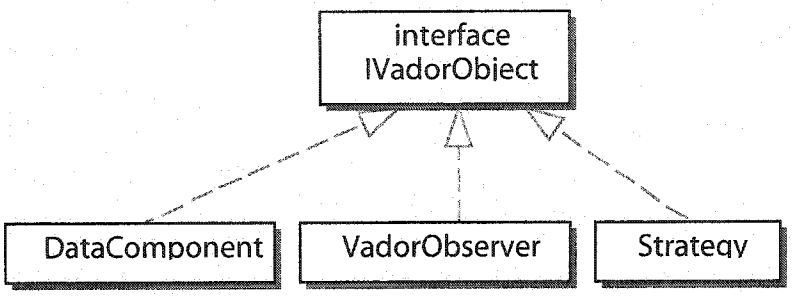

FIGURE 5.10 The Vador Object Module classes diagram

Figure 5.10 shows the class diagram of the Vador Object Module package. This package implements the Vador Object component in the Agent component. The classes in this package work together as the **Composite** pattern and the **Strategy** pattern.

5.7.2.1 Use of the Composite Pattern

Composite Pattern Description The Vador framework accomplishes work by changing and managing the Data and Strategy Components.

The Vador project lets users execute complex process out of simple processes by grouping atomic processes to form larger processes, which in turn can be grouped to form still larger processes. A simple implementation could define classes for process primitives such as Execute a simple program or a command that acts as containers for these primitives.

But there's a problem with this approach: code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. We use the Composite pattern to describes how to use recursive composition so that clients don't have to make this distinction.

The **Composite Pattern** composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

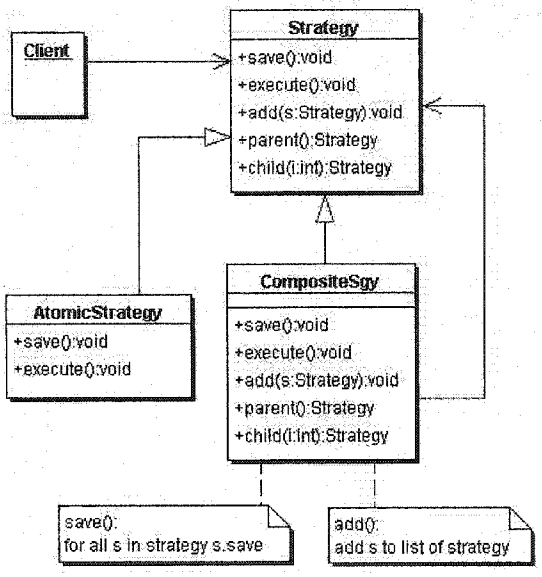

FIGURE 5.11

The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the StrategyComponent object, this class is Strategy. Strategy declares operations like saving and executing that are specific to StrategyComponent objects. It also declares operations that all composite objects share, such as operations for accessing and managing children.

The subclasses AtomicSgy (see preceeding class diagram) defines primitive strategy objects. These classes implement Saving and executing. Since primitive strategies have no child, none of these subclasses implements child-related operations.

The CompositeSgy subclass defines an aggregate of StrategyComponent objects. Because its interface conforms to the Strategy interface, it can compose other CompositeSgy(s) recursively. For instance, it implements saving to call saving on its children, and it implements child-related operations accordingly.

The following diagram shows a typical Composite Pattern for the StrategyComponent objects structure of recursively composed StrategyComponent objects:

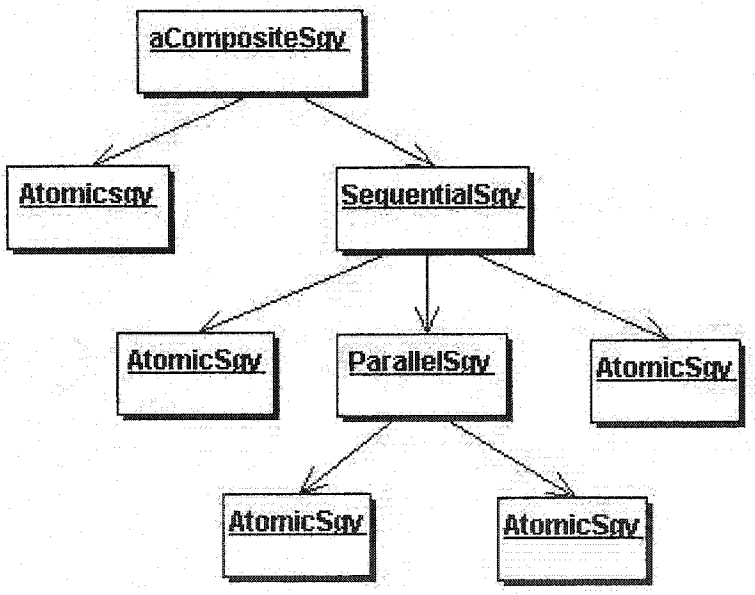

FIGURE 5.12

Participants

- Strategy is a **Component** which declares the interface for objects in the

composition. It implements default behavior for the interface common to all classes. It declares an interface for accessing and managing its child components, and it defines an interface for accessing a component's parent in the recursive structure, and implements it.

- AtomicSgy is a **Leaf** which represents leaf objects in the composition. A leaf has no children and it defines behavior for primitive objects in the composition. The AtomicSgy includes one real program to be executed.
- CompositeSgy is a **Composite** which defines behavior for components having children. It stores child components and implements child-related operations in the Component interface.
- The **Client** (VadorGUI, Librarian Server, Executive Server) manipulates objects in the composition through the Component interface.

Collaborations Clients use the Strategy class interface to interact with objects in the composite structure. If the recipient is a Leaf (AtomicSgy), then the request is handled directly. If the recipient is a Composite (CompositeSgy), then it usually forwards requests to its child components. In some cases, it should perform additional operations before and/or after forwarding.

Consequences The Composite pattern defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.

It makes the client simple. Clients can treat Composite Strategy and individual Strategy objects uniformly. Clients don't know whether they are dealing with an

AtomicSgy or a Composite Strategy. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

The Composite Pattern makes it easier to add new kinds of strategies. Newly defined Composite Strategy or AtomicSgy subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Strategy classes. It makes the Vador project design more general.

5.7.2.2 Use of the Strategy Pattern

Strategy Pattern Description Different processes are encapsulated into Composite Objects, which should be executed in different ways. Some processes should be executed in sequential, or parallel order, but others may be executed in a loop, and use some conditions to determine if they should be continued or not.

Hard-wiring all such processes into the classes that require them isn't desirable for several reasons:

- Clients that need to execute processes get more complex if they include the execution code. That makes clients bigger and harder to maintain, especially if they support different execution methods.
- Different processes will be executed at different times.
- It's difficult to add new execution methods and vary existing ones when processing is a part of a client.

These problems can be avoided by defining new classes that encapsulate different execution methods. A method that is encapsulated in this way is called a strategy.

The **Strategy pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

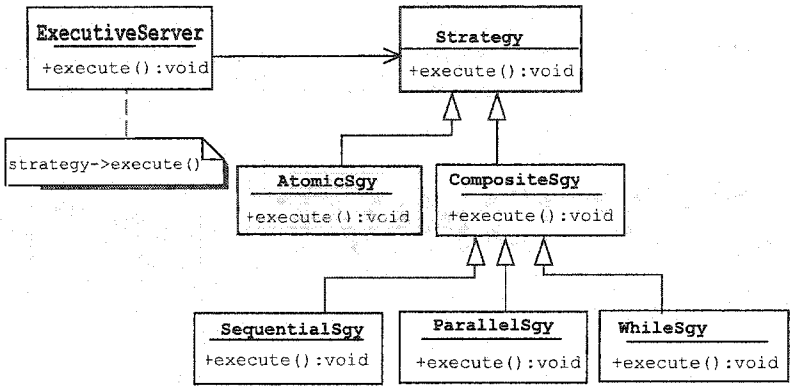

FIGURE 5.13 Strategy class hierarchy

Figure 5.13 illustrates the complete Strategy class hierarchy.

The ExecutiveServer class acts as a client to the class hierarchy, and is responsible for executing the different processes. Execution strategies are not implemented by the ExecutiveServer class. Instead, they are implemented separately by subclasses of the abstract Strategy class. Concrete Strategy subclasses implement different strategies:

- **SequentialSgy** implements a strategy where child processes are executed one after the other in a fixed order.
- **ParallelSgy** implements a strategy where all child processes are executed simultaneously.
- **WhileSgy** implements a strategy that executes its children in a loop.
- **AtomicSgy** includes a single program to be executed.

The ExecutiveServer maintains a reference to a Strategy object. Whenever an ExecutiveServer executes a process, it forwards this responsibility to the corresponding Strategy object.

Participants

- **Strategy** declares an interface common to all supported algorithms. The context (Executive Server) uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy** (SequentialSgy, ParallelSgy, WhileSgy, AtomicSgy) implement the proper algorithms conforming to the Strategy interface.
- **Context** (Executive Server) is configured with a ConcreteStrategy object. It maintains a reference to a Strategy object.

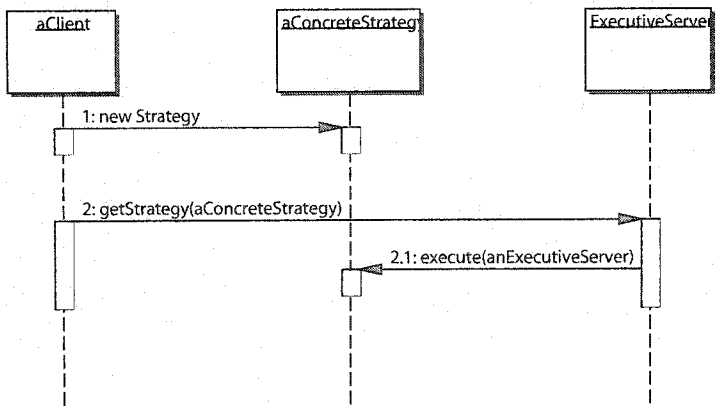

FIGURE 5.14 The Strategy Pattern sequence diagram

Collaborations

- Strategy and Executive Server interact to implement the chosen execution method. An ExecutiveServer passes all data required by the execution method to the strategy when the method is called. Moreover, it passes itself as an argument to the Strategy operations. That lets the strategy call back on the ExecutiveServer as required.

- The Executive Server forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the ExecutiveServer; thereafter, clients interact with the ExecutiveServer exclusively. There is a family of ConcreteStrategy classes for a client to choose from.

Consequences The strategy pattern provides a hierarchy of Strategy classes which define a family of algorithms or behaviors for contexts to reuse. Inheritance helps factor out common functionality of the algorithms (execute function in our case). Encapsulating the algorithm in separate Strategy classes lets us vary the algorithm independently of its context, making it easier to switch, understand, and extend.

5.7.3 The Visitor module

Figure 5.15 illustrates the class diagram of the Visitor Module package.

The classes in this package use the **Visitor** pattern and the **State** pattern to implement the Vador Visitor component in the Agent component.

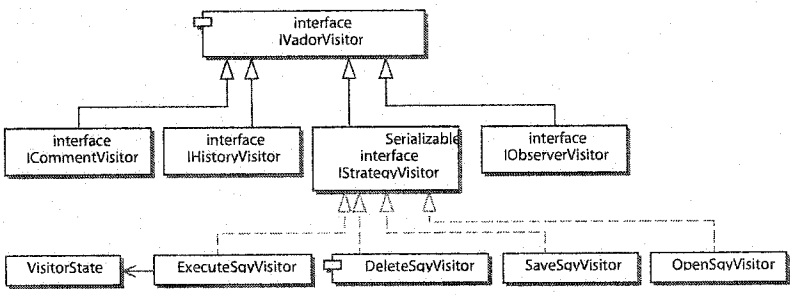

FIGURE 5.15 The Visitor Module classes diagram

5.7.3.1 Use of the Visitor Pattern

Visitor Pattern Description The Vador project represents processes as StrategyComponents, which form a type of abstract tree. Operations need to be performed on the abstract trees such as executing a strategy, saving a strategy in the database. So one might define operations for execute-strategy, save-strategy, and so on.

Most of these operations need to distinguish between StrategyComponents that represent sequential processes from StrategyComponents that represent parallel order processes and so on. Hence there will be one class for sequential StrategyComponent, another for parallel StrategyComponent, and so on (see the Strategy Pattern described before).

For example, a SequentialSgy object on the Librarian Server should perform saving and opening tasks. When it is on the VadorGUI, it should display itself in the graphic view, in a way that lets the user know this is a sequential strategy. When it is on the Executive Server, it analyzes itself and finds the included AtomicSgy objects, then translates them into a series of commands and sends them in the sequential order to the Wrapper server to execute the real applications.

The ParallelSgy works in the similar way but in parallel mode.

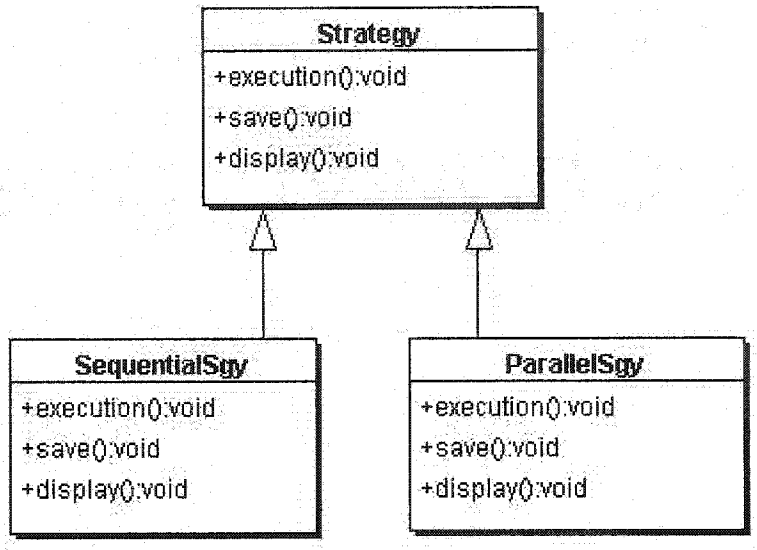

FIGURE 5.16 The strategy class hierarchy

The diagram in Figure 5.16 shows part of the Strategy class hierarchy. The problem here is that distributing all these operations across the various strategy node classes would lead to a system that will be hard to understand, maintain, and change. Confusion will arise from having executing Strategy code mixed with saving Strategy code or opening Strategy code. Moreover, adding a new operation will usually require recompiling all of these classes. It would be better if each new operation could be added separately, and the Strategy classes were independent of the operations that apply to them.

We can have both by packaging related operations from each class in a separate object, called a visitor, and passing it to Strategy elements of the abstract syntax tree–Strategy tree as it is traversed. When a Strategy "accepts" the visitor, it sends a request to the visitor that encodes the Strategy element's class. It also

includes the Strategy element as an argument. The visitor will then execute the operation for that Strategy element, the operation that used to be in the class of the Strategy element.

The **Visitor pattern** represents an operation to be performed on the elements of an object structure. It allows to define a new operation without changing the classes of the elements on which it operates.

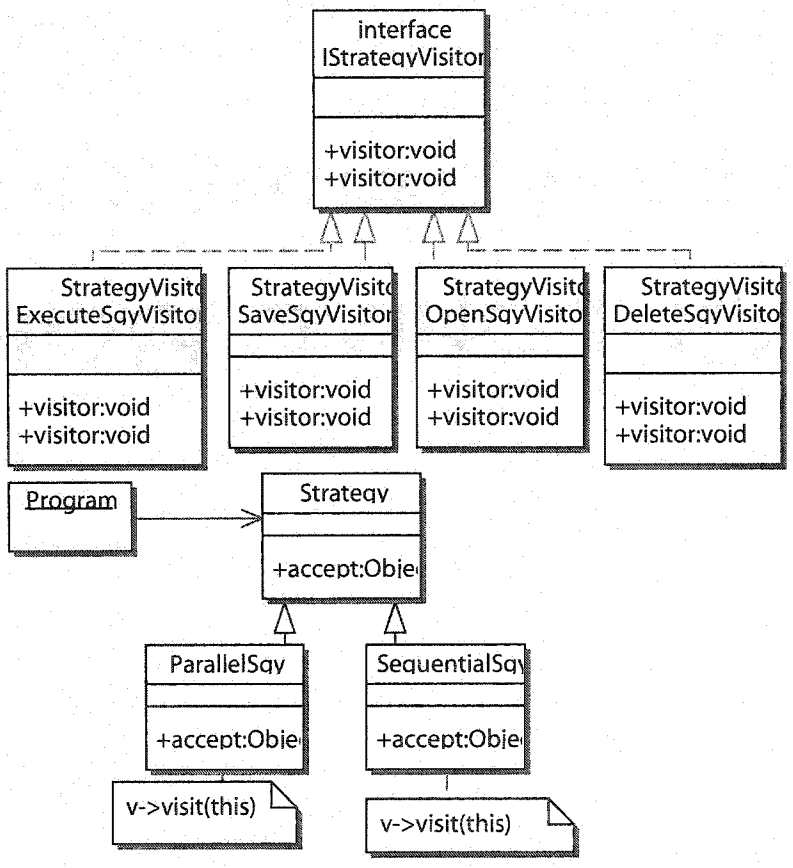

FIGURE 5.17

A Strategy that didn't use visitors might do a saving procedure by calling the saveStrategy operation on its abstract tree. Each of the nodes would implement saving procedure by calling saveStrategy on its components. If the Strategy per-

forms the saving procedure using visitors, then it would create a SaveVisitor object and call the accept operation on the Strategy abstract tree with that object as an argument. Each of the Strategies would implement accept by calling back on the visitor: a SequentialSgy calls the visit operation with a SequentialSgy argument on the visitor, while a ParallelSgy calls a visit operation with ParallelSgy. What used to be the saving operation in class SequentialSgy is now the visit operation on SaveVisitor.

To make visitors work for more than just saving, we need an abstract parent class IStrategyVisitor for all visitors of the abstract tree. IStrategyVisitor must declare an operation for each Strategy class. An application that needs to delete a Strategy will define new subclasses of IStrategyVisitor and will no longer need to add application-specific code to the Strategy classes. The Visitor pattern encapsulates the operations for each saving phase in a Visitor associated with that phase.

With the Visitor pattern, we define two class hierarchies: one for the elements being operated on (the Strategy hierarchy) and one for the visitors that define operations on the elements (the IStrategyVisitor hierarchy). We create a new operation by adding a new subclass to the visitor class hierarchy. As long as the grammar that the Strategy accepts doesn't change (that is, we don't have to add new Strategy subclasses), we can add new functionality simply by defining new IStrategyVisitor subclasses.

This is an acceptable assumption in the context of the VADOR framework, since adding new types of Strategies is much less frequent than adding new operations on the existing Strategies.

Participants

- **IStrategyVisitor** declares a Visit operation for each class of ConcreteStrategy in the object structure. The operation argument type identifies the class that sends the visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
- **ConcreteVisitor** (SaveVisitor, ExecuteVisitor) implement each operation declared to operate on Strategy. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Strategy** defines an accept operation that takes a visitor as an argument.
- **ConcreteStrategy** (SequentialSgy, ParallelSgy) implements an accept operation that takes a visitor as an argument.
- **ObjectStructure** (Program), can be any Vador Server or GUI, or other object that provides a high-level interface to allow the visitor to visit its elements.

Collaborations

- A client creates a ConcreteVisitor and traverse the Strategy structure, visiting each Strategy with the visitor.

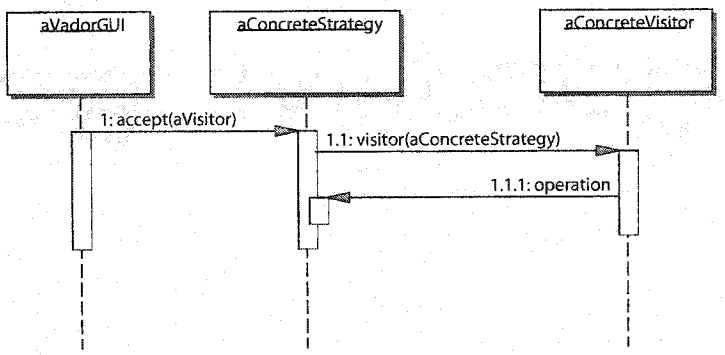

FIGURE 5.18 The Visitor Pattern sequence diagram

- When visiting a Strategy, this Strategy calls the Visitor operation that corresponds to its class. The Strategy adds itself as an argument to the operation to let the visitor access its state.

Consequences The Visitor pattern makes adding operations easy. It gathers related operations and separates unrelated ones. It improves extensibility and flexibility.

5.7.3.2 Use of the State Pattern

State Pattern Description When a StrategyComponent is on the Executive Server, we use the Visitor pattern to execute it (see Visitor Pattern before). The execution (visit) can be in one of several different states: sequential state, parallel state, while(loop) state, atomic state, and so on. The StrategyComponents execution depends on which state it is in, if it is in sequential state, it should execute the next strategy component after finishing the execution of the current strategy

component; if it is in the parallel state, it should wait for all strategy components in the same level as the current strategy component to complete. We use the State Pattern to manage this behavior.

The **State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The key idea in this pattern is to introduce an abstract class called VisitorState to represent the states of the Execution. The VisitorState class declares an interface common to all classes that represent different operational states. Subclasses of VisitorState implement state-specific behavior. For example, the classes SequentialVisitorState and ParallelVisitorState implement behavior particular to the sequential and parallel states of execution.

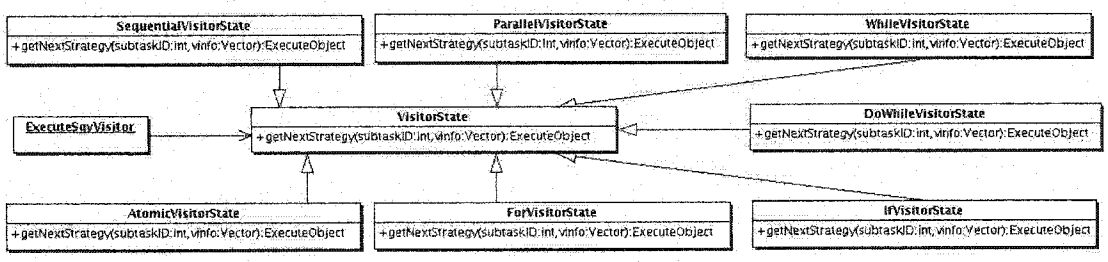

FIGURE 5.19 The classes in the State Pattern

The class ExecuteSgyVisitor maintains a state object (an instance of a subclass of VisitorState) that represents the current state of the execution. The class ExecuteSgyVisitor delegates all state-specific requests to this state object. ExecuteSgyVisitor uses its VisitorState subclass instance to perform operations particular to the state of the execution.

Whenever the execution changes state, the ExecuteSgyVisitor object changes the state object it uses. When the execution goes from sequential to parallel, for

example, ExecuteSgyVisitor will replace its SequentialVisitorState instance with a ParallelVisitorState instance.

Participants

- ExecuteSgyVisitor is the **Context** which defines the interface of interest to clients and maintains an instance of a ConcreteState subclass that defines the current state.
- VisitorState is the **State** which defines an interface for encapsulating the behavior associated with a particular state of the Context.
- SequentialVisitorState, ParallelVisitorState and WhileVisitorState are the **ConcreteState subclasses.** Each subclass implements a behavior associated with a state of the Context.

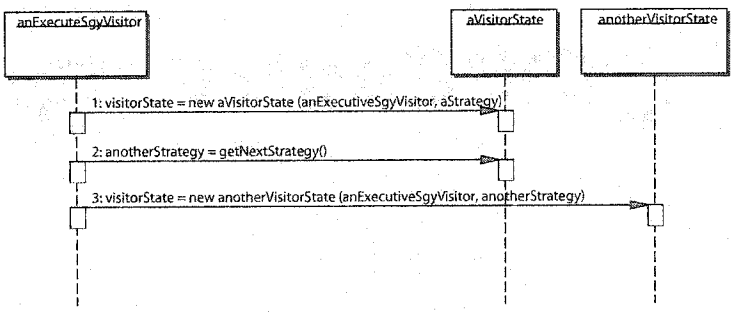

FIGURE 5.20 The State Pattern sequence diagram

Collaborations

- ExecuteSgyVisitor delegates state-specific requests to the current ConcreteState object.

- ExecuteSgyVisitor passes itself as an argument to the State object handling the request. This lets the State object access the ExecuteSgyVisitor if necessary.
- ExecuteSgyVisitor is the primary interface for clients (in our case the ExecuteTask class). Clients can configure an ExecuteSgyVisitor with State objects. Once an ExecuteSgyVisitor is configured, its clients do not have to deal with the State objects directly.
- The ConcreteState subclasses decide which state succeeds to another and under what circumstances.

Consequences The State pattern has the following consequences:

- It localizes state-specific behavior and partitions behavior for different states. The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.
- It makes state transitions explicit. When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal states.

5.7.4 The Proxy module

Figure 5.21 presents the class diagram of the Proxy Module package.

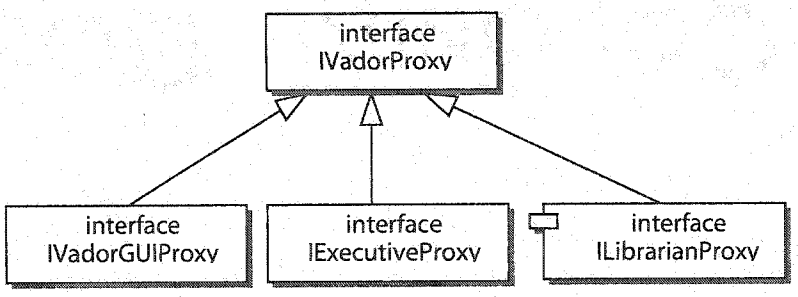

FIGURE 5.21 The Proxy Module classes diagram

The classes in this package use the **Proxy** pattern to implement the Vador Itinerary component.

5.7.4.1 Use of the Proxy Pattern

Proxy Pattern Description Servers lie in the application domain layer. They are the Librarian, the Executive and the Wrapper Servers. We use proxies to access these servers. In this context, the advantage of using a proxy is to provide a local representative for the Server object in a different address space. The client application can treat the remote Servers as if they were locally located through the proxy. This cuts the dependency between the Client applications and the Servers. We use the Proxy pattern to implement the proxy.

The **Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.

To allow the use of a proxy as a Server reference, we need an abstract Server class which defines the interface for both the proxy and the real server, its concrete subclasses (concrete proxy classes and concrete server classes) implement this interface. The Client (VadorGUI, other Vador servers) accesses the remote Server

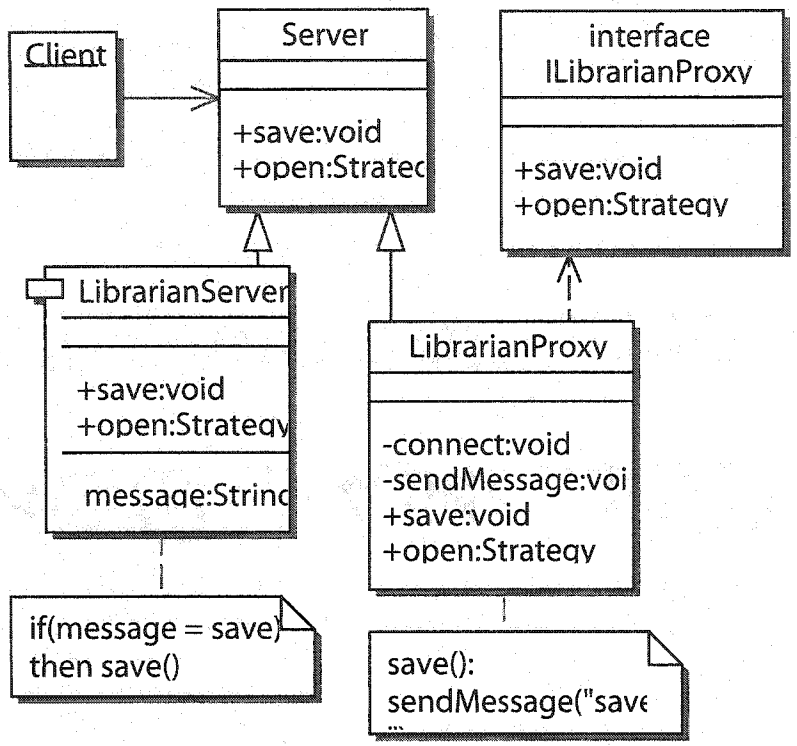

FIGURE 5.22

through the interface defined by the abstract Server class.

For instance, when a Client needs to save a Strategy object into the Database, it invokes the save operation on the LibrarianProxy, which maintains the Librarian Server address as a reference to the Librarian Server. This operation makes a connection to the remote Librarian Server, and then sends a "saving strategy agent" object to the Librarian Server. This triggers the real save function on the Librarian Server to store the strategy object in the Database.

Participants

- LibrarianProxy, VadorGUIProxy and ExecutiveProxy are **Proxies** which

maintain a reference (Server address) that lets the proxy access the real Server. The Proxy refers to a real Server since their interfaces are the same. It controls access to the real Server and is responsible for encoding a request and its arguments and for sending the encoded request to the real Server in a different address space.

- **Server** defines the common interface for Real Server and Proxy so that a Proxy can be used anywhere a Real Server is expected.

- LibrarianServer, VadorGUIServer and ExecutiveServer are Real Servers which define the real objects that the proxies represent. The VadorGUIServer keeps a VadorGUI object pointer which allows to call the functions in the VadorGUI object. The LibrarianServer keeps a connection object which can establish the communication between the LibrarianServer and Database. The ExecutiveServer keeps a list of the running Wrapper Server addresses which can be used to connect to the Wrapper Server objects.

Collaborations The Proxy forwards requests to the Real Server when the client asks.

Consequences The Proxy pattern introduces a level of indirection when accessing a Server. It hides the fact that the Server resides in a different address space, and simplifies the client. Clients can treat the remote Server as a local object. Clients don't know whether they're dealing with a remote object or a local object.

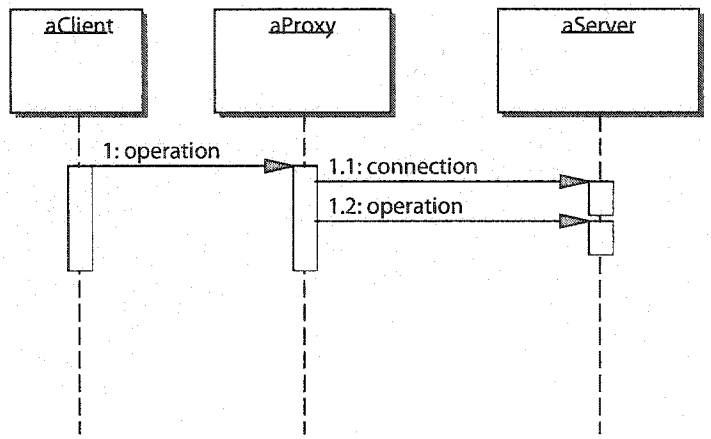

FIGURE 5.23 The Proxy Pattern sequence diagram

5.7.5 The Concrete Agent module

The Concrete Agent modules include the Comment Agent, the Strategy Agent, the Observer Agent and the History Agent module.

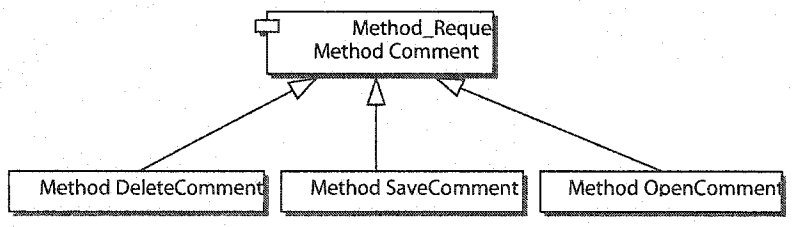

FIGURE 5.24 The Comment Agent module class diagram

Figure 5.24 presents the class diagram of the Comment Agent module package.

The classes in this package implement the Agent behavior of the VadorComment object (the VadorComment object is one of the Vador Objects).

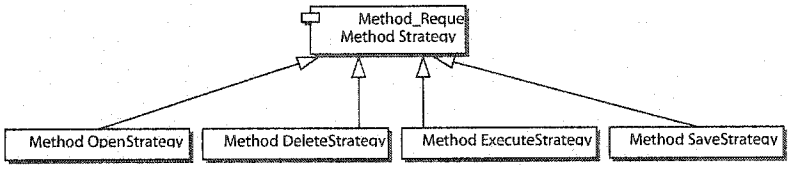

FIGURE 5.25 The Strategy Agent module class diagram

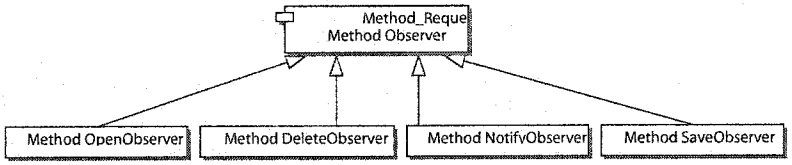

FIGURE 5.26 The Observer Agent module class diagram

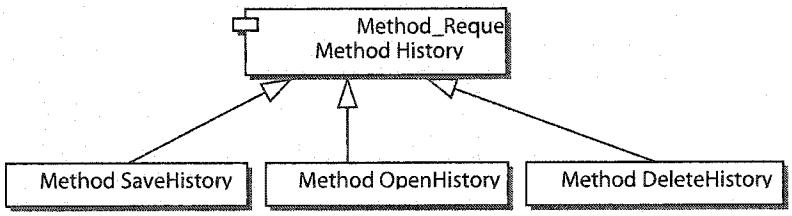

FIGURE 5.27 The History Agent module classes diagram

Similarly, Figure 5.25, 5.26 and 5.27 respectively present the class diagrams of the Strategy Agent, the Observer Agent and the History Agent module package.

The classes in the packages implement the concrete Agents behavior.

5.7.5.1 Method_Request's abstract sub-classes

These sub-classes are the Method_Comment class (in the Comment Agent module), the Method_Strategy class (in the Strategy Agent module), the Method_Observer

class (in the Observer Agent module) and the Method_History class (in the History Agent module).

They define the general behaviors for the various types of Vador Objects. They include:

1. the concrete Vador object class (Strategy, VadorComment, etc.).
2. the abstract Vador Visitor interface for a special type of Vador Object (IStrategyVisitor, ICommentVisitor etc.).

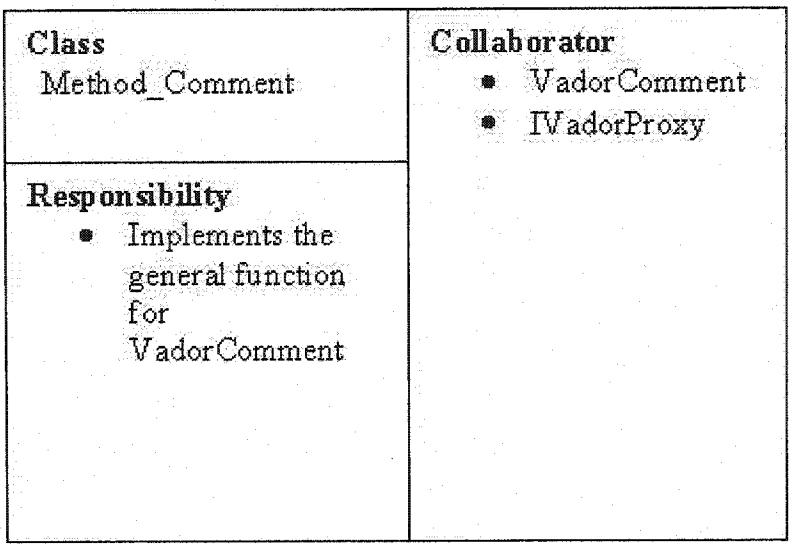

FIGURE 5.28 Method_Comment

An example is given in Figure 5.28 for the Method_Comment.

5.7.5.2 Concrete implementation classes

Each Method_Request's abstract subclass has several concrete subclasses to implement specific behavior of each object. These classes define the concrete methods

that define the concrete Vador Visitor and Vador Proxy for a special Vador Object type. They execute the concrete tasks in the Vador Server. They are the mobile agents.

For example, the Method_SaveComment class is a mobile agent which saves the VadorComment object in the database. It contains the Concrete VadorComment object, the ICommentVisitor and ILibrarianProxy interface. The agent :

1. uses ILibrarianProxy to get the concrete Librarian Proxy from the Server which it is currently in.
2. uses this concrete Librarian Proxy to go to the Librarian Server.
3. dynamically loads the CommentSaveVisitor from the Librarian Server.
4. executes the saving Comment Object task in the Librarian Server.

The Method_OpenComment class is a mobile agent which loads the VadorComment object from the database, it contains the Concrete VadorComment object, the ICommentVisitor and ILibrarianProxy interface, the agent's behavior is almost the same as that described for the Method_SaveComment Agent, except that it executes the opening Comment Object task instead of the saving Comment Object task.

5.7.5.3 Use of Patterns in the Observer Agent Object

The Method_Observer class defines an mobile agent which observes the events occurring in the Vador framework. The classes in the Observer Agent module work together as an **Observer** pattern.

Observer Pattern Description When a Vador object is executed in the Vador framework, it can create different events, such as modification events, execution events, etc.

The Vador users may be interested in some events that occur in the Vador environment. When these events occur, the interested user should be notified automatically.

This behavior implies that the users are dependent on the event objects and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent user objects; there may be any number of different user interfaces to the same event.

The Observer pattern describes how to establish these relationships.

The **Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The key objects in this pattern are subject (event object) and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state.

This kind of interaction is also known as publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Participants

- Event is a **Subject** which defines the event that should be observed.
- **Method_Observer** defines the interface that allows to manipulate the ob-

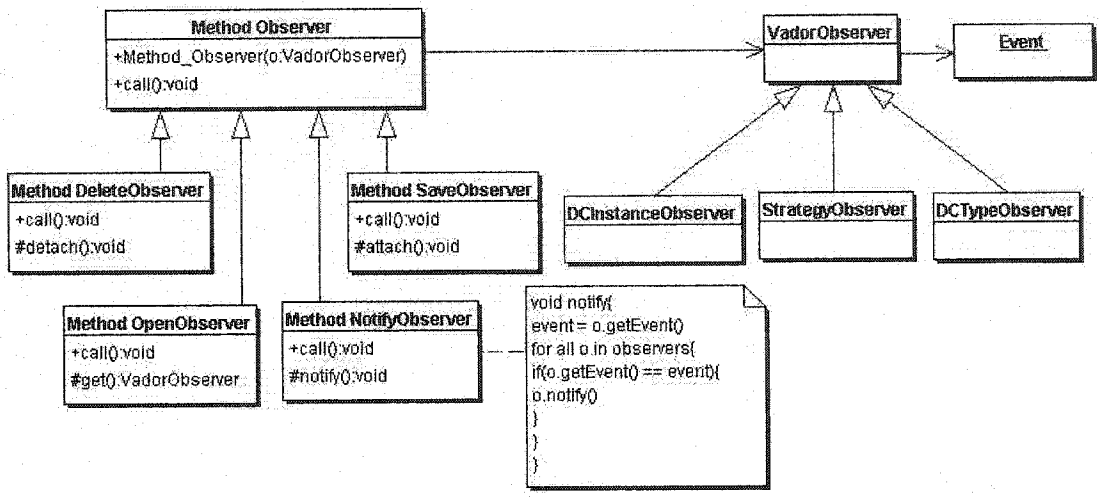

FIGURE 5.29

servers for the events.

- VadorObserver is an **Observer** which defines which user should be notified and which event should trigger this notification.
- Method_SaveObserver, Method_DeleteObserver, Method_OpenObserver and Method_NotifyObserver are the **Concrete Method_Observers** which implement the behavior of attaching, detaching, getting and notifying the Observer objects about the special event. They are the concrete Agents which perform the different concrete tasks.
- DCInstanceObserver, DCTypeObserver and StrategyObserver are the **Concrete Observers** that define which types of events should be observed.

Collaborations

- The Method_SaveObserver Agent registers the ConcreteObserver objects for

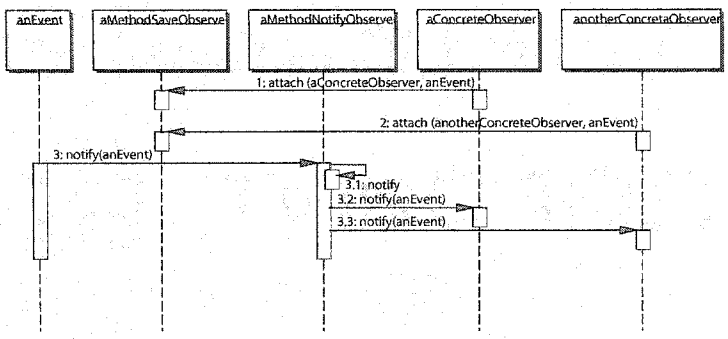

FIGURE 5.30 The Observer Pattern sequence diagram

the events of interest.

- Subject (Event) notifies using the Method_NotifyObserver Agent whenever a change occurs.
- The Method_NotifyObserver Agent uses its registered ConcreteObservers list to find the interested ConcreteObserver objects and inform them.
- After being informed of a change in the subject with the information, the ConcreteObservers (DCInstanceObserver, DCTypeObserver, StrategyObserver) objects use this information to reconcile their state with that of the subject.

Consequences The Observer pattern lets vary events and observers independently. We can reuse event objects without reusing their observers, and vice versa. It lets us add observers without modifying the event objects or other observers.

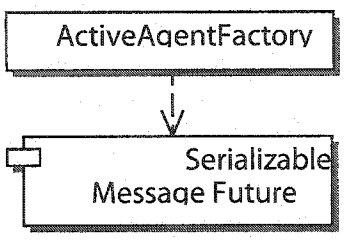

FIGURE 5.31 The Agent Tools module classes diagram

5.7.6 The Agent Tools module

Figure 5.31 presents the class diagram of the Agent Tools module package. The classes in this package are helper classes, they provide the programmer with a convenient way to create and manipulate the Agent component. The Client (VadorGUI Model) creates the Active Agent object through the ActiveAgentFactory class, and gets a Message_Future object when necessary.

- **ActiveAgentFactory** class (Illustrated in Figure 5.32).

  The clients use this class to easily create different concrete Agents.

- **Message_Future** class.

  When a client creates a Active Agent, it receives a Message_Future object. The Message_Future object allows the clients to obtain the result of the Active Agent's execution. Each Message_Future reserves the space to store its result. When a client wants to obtain this result, it can rendezvous with the Message_Future, either blocking or polling until the result is computed and stored into the Message_Future.

Class Message_Future	Collaborator
Responsibility • Stores the result of a method call on an active object • Provides a rendezvous point for a client	

FIGURE 5.32 Message_Future

5.8 Architecture of the Server component

The **Vador Server** is a Java process which supports multi-threading. It provides an execution place to the agent. The Vador Servers are the Librarian Server, the Executive Server, the VadorGUI Server and Wrapper Server. (We will discuss the Librarian Server and the Executive Server in the following section). The Vador Server provides a conceptual and programming metaphor where agents are executed. It also provides a consistent way to define and control access levels, and to control computational resources. It offers several services:

1. agent creation

   Under certain situations, the Vador Server can create an Agent to carry out some task, for example, when a task finishes, the ExecutiveServer creates the ObserverNotify Agents to notify the registered users that the task has completed.

2. agent execution

   When the Vador Server receives the Agent, it calls the Agent's invoke function to activate the Agent's execution.

3. agent mobility.

   The Vador Server provides the platform where the Agent can migrate.

4. access control.

   The Vador Server provides access control functionality, for instance the Agent communicates with the database through the LibrarianServer, and it is the LibrarianServer who decides whether the data in the database can be accessed by the Agent or not.

5. agent persistence.

   The execution of tasks in the Vador system often takes a long time. To avoid the loss of the Agent parameters in the case of an ExecutiveServer's crash, the ExecutiveServer writes the Agent's parameters in a file which is stored on the hard drive before the execution, and this file can be used by the ExecutiveServer to restore the Agent as necessary.

6. communication with agents.

   The Agents and Servers can communicate with each other through standard interfaces.

Both the ExecutiveServer and the LibrarianServer are part of the **active agent pattern**. They are specializations of the generic execution place where a mobile agent can execute its tasks.

Figure 5.33 shows the class diagram of the Librarian Server and Figure 5.34 presents the class diagram of the ExecutiveServer.

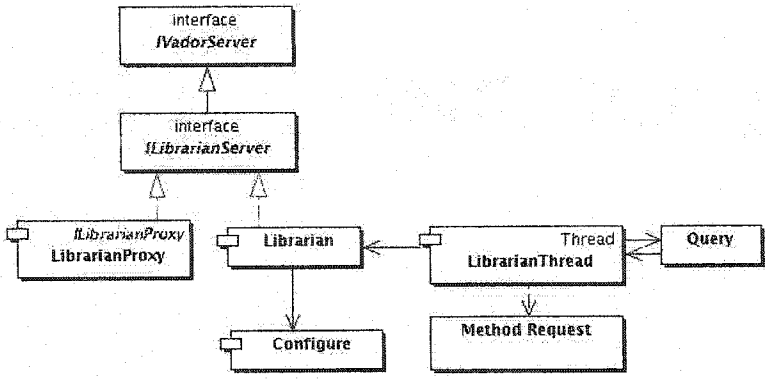

FIGURE 5.33 The Librarian Server Package

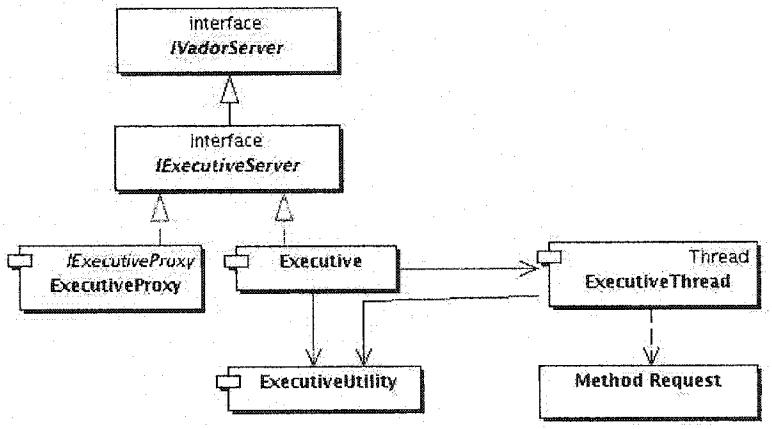

FIGURE 5.34 The Executive Server Package

The main classes are the Librarian class and the Executive class. Their common base class is VadorServer. These two classes listen to a special port and when a request arrives, they create a LibrarianThread object or an ExecutiveThread object which encapsulates a thread and dispatches a ServerSocket object to it. This thread keeps the communication with the client and performs the operations requested. If the arriving object is an Agent (Method_Request's concrete subclass). The thread

invokes the Agent's call operation to start its execution on the Server.

The Vador Client applications or other servers access the Librarian Server (Executive Server) through the LibrarianProxy (ExecutiveProxy).

In the Librarian Server Package, the Configure class defines the parameters which are used by the Librarian Server to connect with the Database; the Query class has the functions which use the SQL language to query with the Database.

The ExecutiveUtilities class in the Executive Server Package defines the tool functions which can be used by the Executive Server to manage the execution tasks and communicate with the Wrapper Server.

5.9 Active Agent Collaboration and Dynamic Behavior

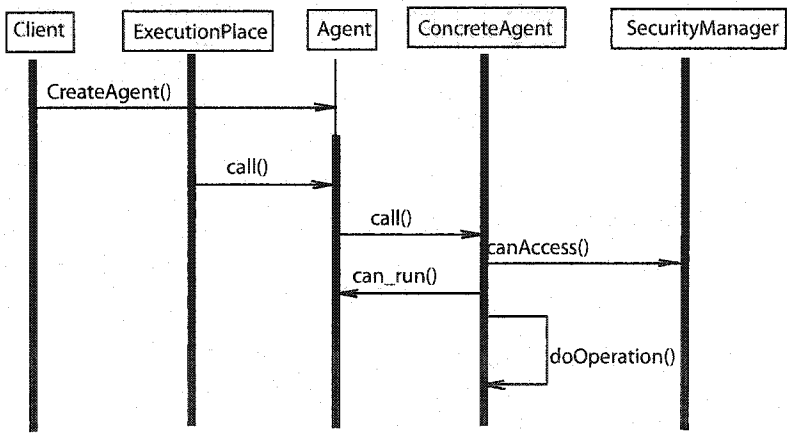

FIGURE 5.35 Interaction of the Agent pattern

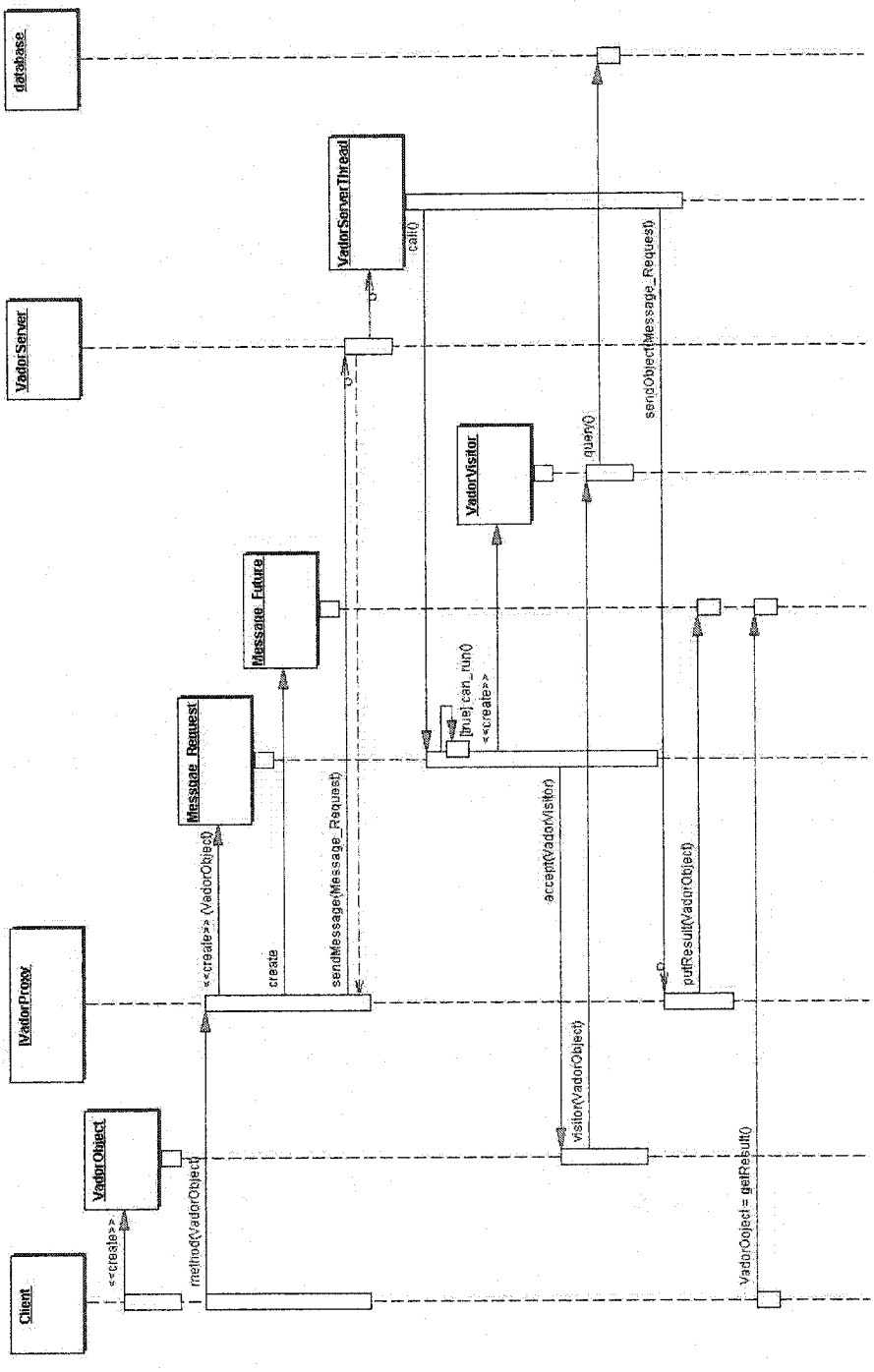

FIGURE 5.36 Active Agent pattern Sequence diagram

In order to accomplish some work using the VADOR framework, clients, such as the VadorGUI, create agents and send them to an execution place (the Vador Server), which calls the standard agent operation (call function). Depending on the agent's security policy, the operation is executed, or not, on the related agent instance.

Figure 5.35 shows the interaction of the Active Agent pattern. The process is detailed further in the diagram presented in Figure 5.36. In Figure 5.36, the ConcreteAgent component is divided into its components, the Message_Request, the VadorObject, the IVadorProxy and the VadorVisitor.

There are three phases in the dynamic behavior of the Active Agent pattern.

1. Agent construction and sending.

   A client invokes a method in the ActiveAgentFactory object. This triggers the creation of the Agent object (Method_Request subclass object), the Agent object includes the concrete Vador Object, the Vador Visitor interface and any other binding required to execute the method and return its result. It also includes the Vador Proxy objects (if there are several Vador Proxy objects in the Agent object, this means the Agent will travel through several Vador Servers) which keep the Vador Server address. The Agent object can include several Vador Proxy objects, this makes the agent travel through several Vador Servers to complete its execution. The Agent object uses the Vador Proxy to send itself to the next Vador Server. If the Agent has a two-way invocation, a Message Future is returned to the client. No Message Future is returned if an Agent is a one-way, which means it has no return value.

2. Agent execution.

   The Agent runs in the Server process. After receiving the Agent object, the server calls its *call* operation to start the execution. The Agent determines

if itself can become runnable by calling its guard method. When it becomes runnable, it dynamically loads the concrete Vador Visitor class and uses it to execute the task and create its result if it is a two-way method invocation. If the Agent needs to go to the next server to complete its execution, it dynamically loads the concrete Vador Proxy class and uses it to send itself to the next Vador Server.

3. Completion.

   In this phase, the result is sent back by the Vador Server and stored in the Message_Future object that is created at the time of creation of the Agent. The client can quote the Message_Future to obtain execution result.

5.10 Consequences and Comparison with Related Work

The use of the Active Agent pattern brings the following benefits to the overall VADOR framework: Benefits of the Active Agent pattern are:

1. Easy development of dynamic and distributed applications.

   The Active Agent pattern defines the standard interfaces of the concrete agent. When we want to develop a new concrete agent to perform a new functionality, we simply implement and extend these standard interfaces, then dynamically deploy them to the whole system as plug-in objects, the existing VadorServers can recognize and use them automatically as they know the agent standard interfaces.

2. Simple integration with definition and management of user.

   Each user in the Vador system has a unique identifier and belongs to a special group. When the user create an agent, his user identifier is included in

the agent and this agent becomes his delegate. The user identifier and his belonging group give the Active Agent pattern an easy way to manage the access, security and priority problems.

3. Flexible definition of agent's security policy.

   The security policy will be discussed in another research topic.

4. Support concurrency, scability and flexibility.

   Each VadorServer can run many concrete agents concurrently using special synchronization rules to share the static resources between them, so the Active Agent pattern supports concurrency.

   As the work group extends, we can increase the number of VadorServers if necessary. For example, we can have a new ExecutiveServers in the framework. In this case, when an agent arrives at an ExecutiveServer, this server can decide to ask the agent to send itself to another ExecutiveServer if it is very busy. These capabilities bringing the scability and flexibility.

The comparison of the Active Agent pattern and other related Agent patterns is presented below:

5.10.1 Agent patterns comparison

Figure 5.2 shows the structure of the Active Agent pattern in the Vador framework. The Active Agent pattern has been developed directly on top of the Java Virtual Machine. The client in the Active Agent Pattern is the Vador application (the VadorGUI).

All the Agent Support System (ASS) is implemented directly in the Vador System. Although this ASS is relatively simple compared with other professional ASS (it

is specifically designed for the purpose of the Vador framework), it has almost all the characteristics that a complete ASS should have, such as mobility, autonomy, some limited sociability and reactivity.

The agent in the Active Agent pattern always works in the dispatching model (the agent pushes itself from its current host to a remote host). There is no communication between agents, and the agent will be destroyed immediately when it finishes its tasks.

The Vador framework doesn't offer tools that allow other developers to create their own mobile agent applications in the Vador framework.

Comparing with the Agent pattern in the AgentSpace framework by Silva and Delgado (2001), AgentSpace provides an extensible and elegant way to handle security policies/strategies related to the access and interactions between agents and end-users, and between agents themselves which is currently being addressed as a separate research topic. The client in the AgentSpace is a Java Applet. Agents always run on some AS-Server's context, they interact with their end-user through applets running in some Web browser's context. The client accesses agents indirectly through the AgentView (the agent proxy).

The AgentSpace has been developed on top of the Voyager infrastructure. It is a Java library that enables developers to create their own mobile agent applications in the AgentSpace.

The Agent pattern in the Aglets Workbench [Silva and Delgado (2001)] has been developed on top of the Java Virtual Machine, there is currently no User and SecurityManager references in the Aglets' agent. The client (applet) accesses agents indirectly through the AgentProxy. The agent can have many behaviors, such as creation, cloning, disposal, dispatching, retracting, activation, deactivation and

messaging (see the detail explanation in the chapter 2). It is a Java library that enables developers to create their own mobile agent applications in the Aglets WorkBench

Comparing the Active Agent pattern and the other two agent patterns, we see that in terms of performance and functionality, the Active Agent pattern is currently a smaller ASS, it is simpler, and has less functionality.

The performance of Active Agent pattern is relatively high, as it works specially in a limited domain (Vador environment), it has less layers and less useless functionalities.

In terms of design structure, we can see that: Advantage of the Active Agent pattern is that it implements the ASS itself. This brings more control over the whole system. In the Active Agent pattern, one can only access agents through a VADOR compatible applications (including maybe a web browser, see discussion in the chapter 7). This gives us far more control over the evaluation of the framework, and enhanced possible scabilities to implement proprietary security policies. Disadvantage coming from the fact that the Active Agent pattern implements the ASS itself is that we have more works to do and more security problems to manage.

5.10.2 Conclusion

The Active Agent Pattern implements the Agent Support System (ASS) itself. This gives us the flexibility to improve and extend our system easily. Its design structure is good, but from the view of the performance and functionality, we can see it has a number of shortcomings as compared with more general environments:

1. the communication system between the agents and servers is very simple;

2. the agent's anti-attack and recovery abilities are low;

3. the security policies/strategies are not implemented yet.

As the Vador Agent System environment is a complete ASS environment, we believe that all these shortcoming can be resolved as the system evolves.

The following chapter will use an example to show how to use the Vador framework to execute a concrete task, and the last chapter will discuss how easy it is to add new functionalities in the Vador framework.

CHAPTER 6

MANAGING AN OPTIMIZATION PROCESS USING VADOR

This chapter presents an example to show how to accomplish actual work and manage execution results using the Vador framework.

6.1 Airfoil shape optimization, a step toward MDO

6.1.1 Optimization Problem

The design of planes goes through several stages and involves many disciplines, individuals, computers, and organizations. The objective of this example is to illustrate several aspects encountered during the use of the Vador Framework to complete an optimization process of wing sections, without paying much attention to the underlying logic of the industrial design process.

In a wing section design process, the number of design variables is directly dependent on the geometric representation and parameterization of the wing. A geometric representation of a two-dimensional airfoil that only requires a few parameters can greatly simplify the design process. NURBS (Non Uniform Rational B-Spline) can be used to develop good approximations of airfoil shapes [Trépanier *et al.* (2000)]. The NURBS representation can later on be used as input to analyze the aerodynamic characteristics of the section. The example discussed here describes a process which aims to search for an optimum NURBS representation of a wing section using an optimizer, which relies on an objective function that in-

cludes only geometric criteria, but which is submitted to constraints from various disciplines which fixes a minimum tolerance for the approximation.

6.1.2 Methodology to obtain an airfoil approximation

Because a NURBS is a curve that aims to fit a given set of points representing a profile, one must be able to evaluate the adequacy of the NURBS geometric representation compared with other less convenient means of representation of the section. Once an approximation error is defined, a NURBS that adequately minimizes this error is sought. The approximation error is calculated using a combination of the maximum distance and the average distance between the target curve and the current approximation(cf. Fig. 6.1).

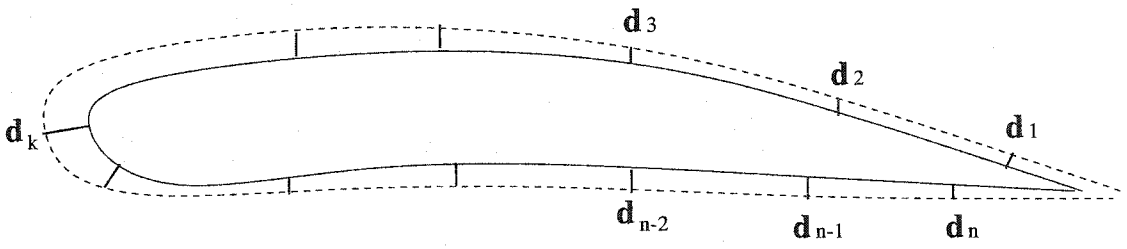

FIGURE 6.1 Evaluation of the approximation error

The methodology to obtain a geometric approximation using a NURBS proceeds as follows:

- create a target curve using a B-spline with as many control points as there are points in the discrete profile; such an interpolating spline is obtained based on the given original discrete points, the corresponding knot vector and by solving a tridiagonal linear system.

- select the number of control points M, the order of the NURBS $P+1$, and the number of points n upon which is calculated the approximation error (different than the number of points in the discrete profile);
- an initial guess for the NURBS is obtained by concentrating the knots according to the curvature of the profile, and by considering a non-uniform B-spline (i.e. a NURBS with ω_i=1);
- an optimization algorithm (quasi-Newton BFGS)is used to refine the geometric representation of the objective airfoil profile until the error tolerance is reached.

6.2 Implementation of the example in the Vador Framework

6.2.1 Analysis of the example

Figure 6.8 shows the flow chart of the geometric optimization process of wing profiles. We can divide the process into two phases: the initialization phase and the loop phase:

1. **The initialization phase** creates the target curve.

   The application `inter.exe` receives two input files – `Para_inter.dat` and `Profile_air` to create a target curve. This curve will be stored in the file `profile_inter.pie`. As the `Para_inter.dat` and `Profile_air` files are both created through external processes, we call these external inputs.

2. **The loop phase** creates the approximation curve and compares it with the target curve. If the distance between them is smaller than the target approximation error, the loop will be stopped. If not, the loop continues, to

create a new approximation curve, and compute a new distance to see if the tolerance is satisfied. This process will be repeated until the allowed distance or the maximum number of loop is reached.

We divide this phase into several steps:

(a) Calculating the number of control points.

(b) Initializing the new control points.

(c) Calculating the distance between the target curve and the approximation curve.

(d) Comparing the allowed distance with the computed distance.

(e) Checking whether the allowed distance or the maximum loop number has been reached.

6.2.2 Design of the example in the Vador Framework

Here are the main steps needed to define the relevant objects in the framework in order to manage the optimization process using Vador. A complete discussion on how the VADOR framework may be used can be found in Appendix II.

1. Define the DCType objects. The Vador Framework uses the DCType objects to describe the data types produced by each process. Different data types use different DCType objects. for example, the `profile_inter` object describes a data type used by the `inter.exe` program and its output file has an extension "pie" (See Figure 6.2).

2. Define the StrategyComponent objects.

   The Vador Framework uses StrategyComponent objects to describe the processes.

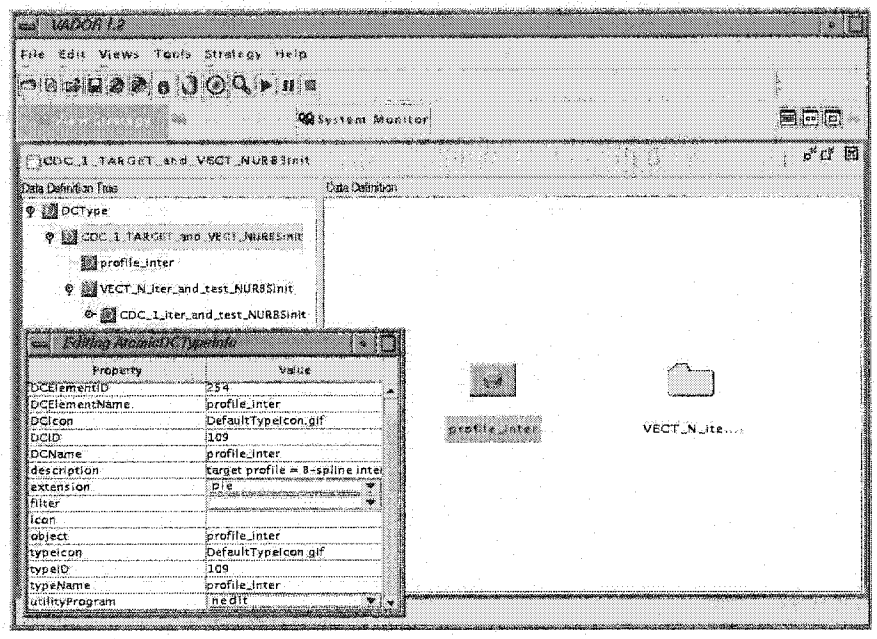

FIGURE 6.2 Displaying the DCType objects in the VadorGUI

The example process can be divided into two processes which are executed in sequential order, and the second process is a loop process which includes several small processes which are executed sequentially. So we use the Composite pattern to define the composite strategy, and use the Strategy pattern to define the different types of processes, for example the SequentialSGY is used to define sequential processes, the DoWhileSGY is used to define the loop process, and the AtomicSGY defines the small processes which include the legacy application such as `inter.exe`, `Iter_nb_pt_ctrl.sh` and so on.

3. Define the DCInstance objects.

   The Vador Framework uses the DCInstance objects to describe the concrete execution processes.

   After the definition of the DCType and StrategyComponent objects, we

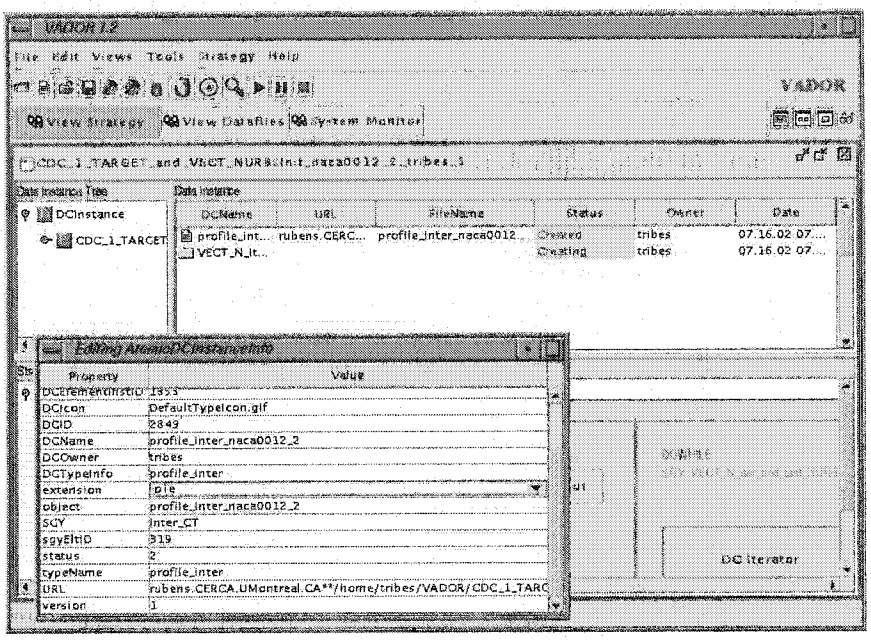

FIGURE 6.3 The VadorGUI shows the status of the DCInstance objects creation

should define the DCInstance objects. These objects define the concrete data and concrete process parameters which will be used during the execution. They are based on the DCType and StrategyComponent objects. Figure 6.3 presents the Vador interface as instances are being created.

4. Using the VadorGUI

   The Vador Framework provides a graphical user interface that lets users create and manipulate interactively their own Data and Strategy Components;

   Figure 6.4 shows a debug viewer which displays the execution information in the VadorGUI. The user can use this information to determine where the errors have occurred, in case of execution problem.

5. Saving the Vador objects in the Database

   The Vador Framework uses the Librarian Server to interact and manipulate

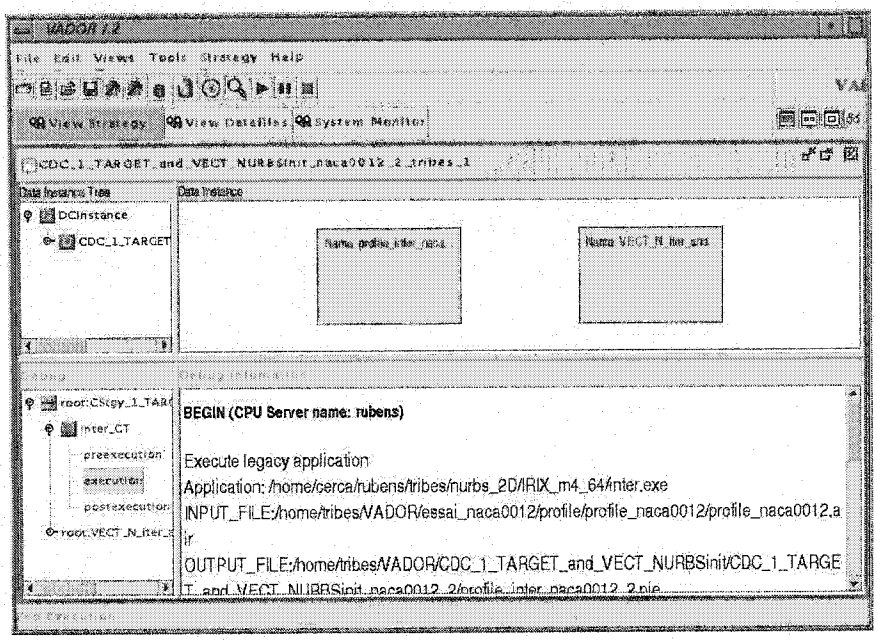

FIGURE 6.4 Debug viewer in the VadorGUI

with the database, saving the Vador objects in or loading the Vador objects from the Database.

6. Executing the processes

   The Vador Framework provides the Executive Server which is responsible for the execution of the processes, and for sending back execution results to the Librarian Server when an execution step has completed.

   Figure 6.3 shows the application `inter.exe` that has been executed and its object profile_inter_naca0012_2 that has been created. The VadorGUI has received the success notification and reflected this notification in the GUI.

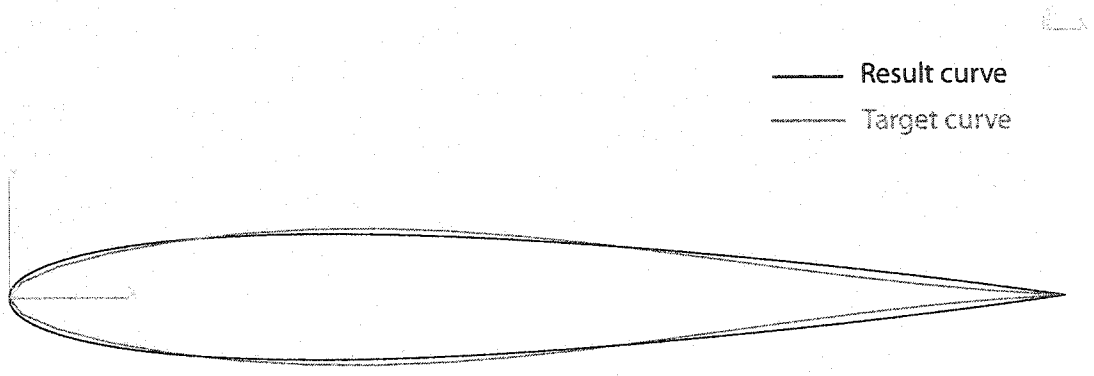

FIGURE 6.5 The Execution Result

6.3 Result of Testing the example in the Vador Framework

Figure 6.5 shows the resulting curves obtained through a sample optimization process execution. The black curve is the result curve (the result data is stored in the `profile_init.pie`), the gray curve is the target curve (the target data is stored in the `profile_inter.pie`). We can see a clear difference between the two as the error tolerance is set relatively high. In practice, these two curves can be brought very closely together when the tolerance is reduced.

To illustrate the behavior of this process as it is run using the VADOR framework, several executions have been performed for a range of initial conditions. Each run corresponds to an optimization process initialized with an increasing number of control points on the NURBS. As the initial number of control points in increased, the approximation problem becomes simpler, and execution time and traffic volume decrease.

To illustrate the execution performance, two aspects are presented: execution time and transaction volume. The execution time is computed as the elapsed time

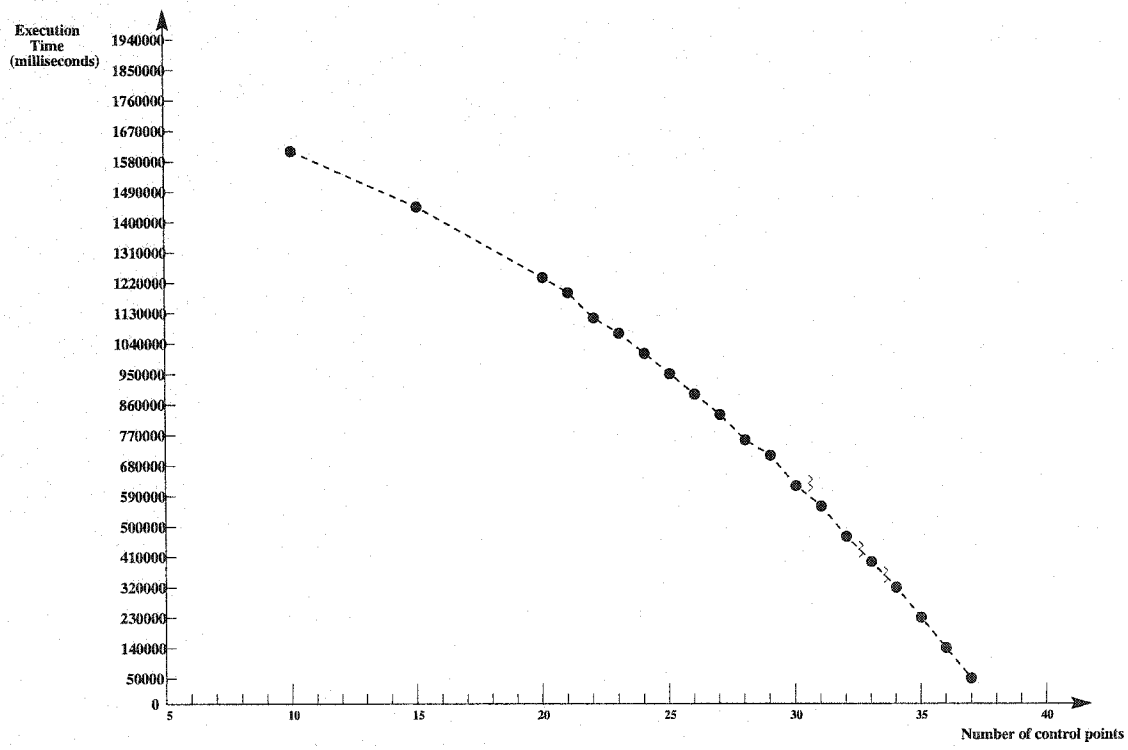

FIGURE 6.6 Execution time

between the beginning execution time and end execution time. The transaction volume is the amount of data that is transfered among the machines during the

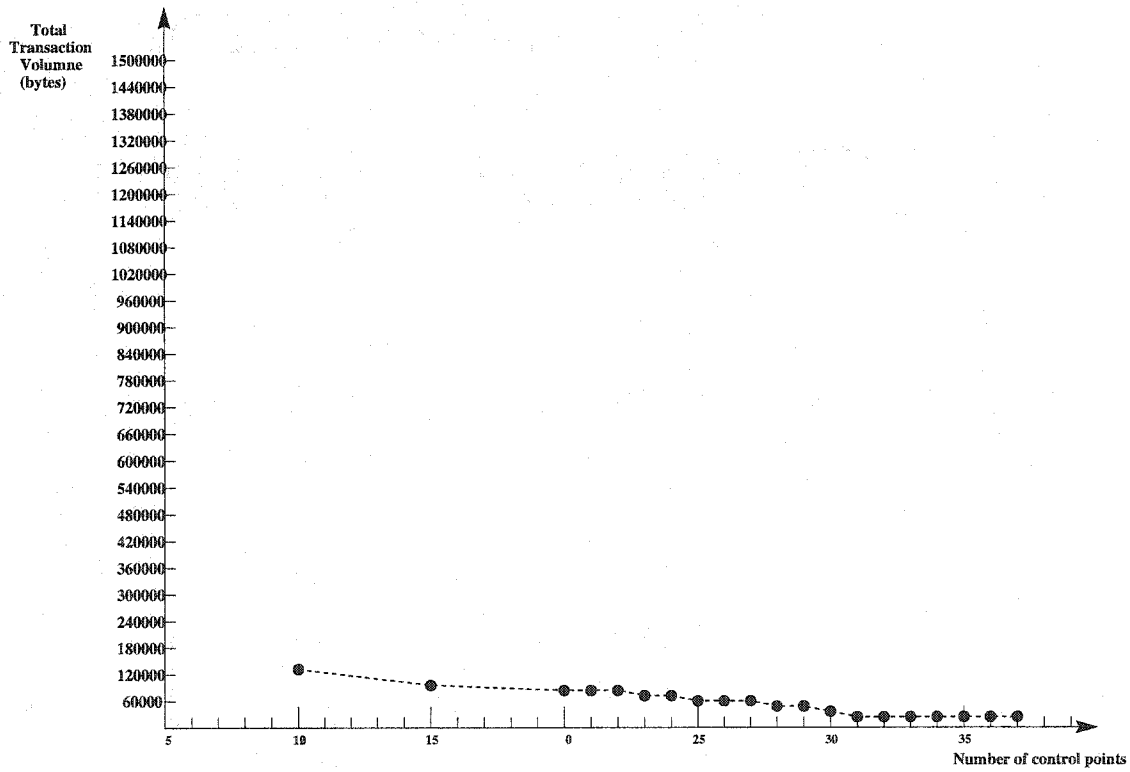

FIGURE 6.7 Traffic volume

execution period. Both Execution time and transaction volume directly contributes to the perceived overall efficiency of the system, and constitutes some of the most important concerns of users.

To get the correct result, it is not easy to get pure testing results using a public network since there are many other applications running on the computers concurrently. Those applications consume resources of the machines and may exchange data through the network; the testing environment thus always changes, and execution time and transaction volume data can not be obtained accurately.

Therefore, to obtain precise test results, five personal computers have been connected together to form a small isolated local network on which all tests were run. Among these machines, one acted as a server and provided naming services and connection to the external network. It was identified as the "master". The other four machines were identical nodes of the network, named "node01" to "node04".

Figure 6.6 and Figure 6.7 show the execution performance of the testing example using the VADOR system. The detailed test description can be found in Zhou 2003.

As this example demonstrates, the VADOR framework can be used to integrate and automatically execute complex analysis and optimization processes, and manage the results produced.

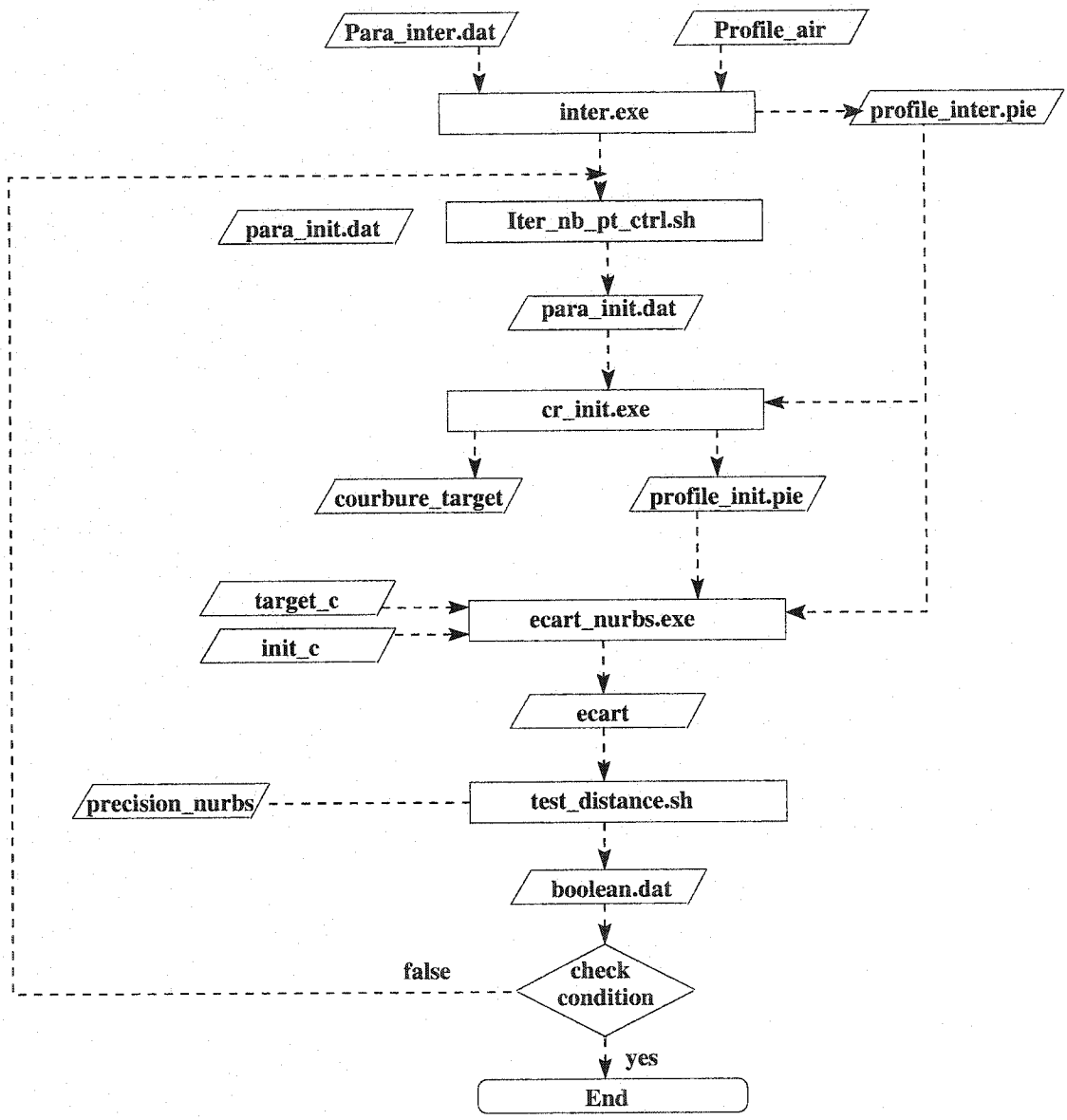

FIGURE 6.8 Geometric Optimization process of wing profiles

CHAPTER 7

EXTENDING THE FRAMEWORK: A CASE STUDY

As mentioned in the opening of chapter 3, using design patterns as the architectural basis of Vador should allow us to easily extend the framework. This chapter presents a discussion of the main extension points of the framework and an example to illustrate how to extend the Vador framework. This chapter is divided in two parts. First, the main extension points provided for in the VADOR architecture are summarized and discussed, and second, a short case study is presented, that highlights the main steps needed to extend the framework by adding a new service.

7.1 The Extension Points in VADOR

Extension points are grouped in three parts:

- Extension Points in the VadorGUI Module.
- Extension points for other clients.
- Extension Points in the domain layer.

7.1.1 Extension Points in the VadorGUI Module

7.1.1.1 New menu displaying status through the Mediator Pattern

We can easily have a menu displaying status (such as the menu is enabled or disabled, selected or unselected, etc) by extending the VadorGUIDirector class in the Mediator Pattern to get a new concrete Director class, and use this concrete Director class with some special objects displayed in the VadorGUI to get the new menu displaying status.

7.1.1.2 New DCViewers and StrategyViewers using the Template Method Pattern

New viewers can be easily added in the VadorGUI through extending the base classes such as DCViewer and StrategyViewer class in the Template Method Pattern.

7.1.2 Extension points for other clients

Although we normally access the VADOR system through the VadorGUI application, we can also access it through other forms of client applications by using the Server's proxy class. For example, the Loadbalancing application (see Liu (2003)) uses a batch client which directly accesses the Executive server through the Executive proxy class. A second example of such client extension is discussed in the next section.

7.1.3 Extension Points in the domain layer

There are several extension points in domain layer. They are located in the participants of the Active Agent pattern: agent, execution place and concrete agent. The five most important points are presented here.

7.1.3.1 New Agents based on the Command Pattern

New agents can be easily introduced in the VADOR system by simply implementing the IActive_Agent interface which is defined by the Command Pattern in the Active Agent pattern (see 5.7.1.1).

7.1.3.2 New composite objects using the Composite Pattern

Based on the Composite Pattern, we can agglomerate objects and treat them as one object. For instance, by extending the CompositeSgy class, we can introduce new composite strategies. The same is also true for DataComponents. These extension points form the core classes of the VADOR data module.

7.1.3.3 New Strategies using the Strategy Pattern

There is a set of the concrete Strategies in the VADOR system, such as the SequentialSgy, the ParallelSgy, the WhileSgy and so on. We can introduce new forms of concrete Strategies by extending the Strategy class.

7.1.3.4 New operations on Strategies using the Visitor and State patterns

Each concrete strategy object implements different operations in the different execution places. These operations can be enhanced through the new concrete Visitor class which implements the IVadorVisitor interface in the Visitor Pattern and a new concrete State class which extends the VisitorState class in the State Pattern.

7.1.3.5 New proxy on the Proxy Pattern

The Agent object accesses the Servers through the Proxy. Each server has its proxy. By implementing IVadorProxy, a new Proxy object can be inserted in the framework to bring the functionality of a new Server.

7.2 Adding a Web Service

We now suppose that the Vador system needs to provide a new service – the web service. This service receives requests from a web browser and sends back results after dealing with these requests. Now let's see how we can extend the Vador system to implement this new functionality.

The new agent implements the standard Active Agent interface. Existing Vador Servers do not even need to be restarted to use this new agent object as they know how to interact with the standard Active Agent interface.

The whole Vador system is implemented as an **Active Agent pattern**. A new Vador Server (the VadorWebServer) and the new Agents (the VadorWebAgent) can be introduced in the system to implement the web service functionality.

7.2.1 New Vador Server – the VadorWebServer

The VadorWebServer is a new server which provides a platform for the agents to communicate with the web browser. It takes in charge the responsibility of accessing control.

To let the Agent work on the Server, the VadorWebServer should implement two functionalities.

- Invoking the agent

  The VadorWebServer should call the Agent's "call" function to activate the Agent after its arrival.

- Mobility

  The VadorWebServer should carry out the mobility functions such as receiving and sending the Agents, these extending points are found in the execution place.

7.2.2 New Agent – the VadorWebAgent

As described in the chapter 5, an Agent in the Vador System has three components. They are the Vador Object, the VadorVisitor and the VadorProxy.

7.2.2.1 Components of the VadorWebAgent

- **WebService – VadorObject**

  The WebService abstract class implements the IVadorObject interface. It has a concrete subclass – the WebRequest class which includes the request

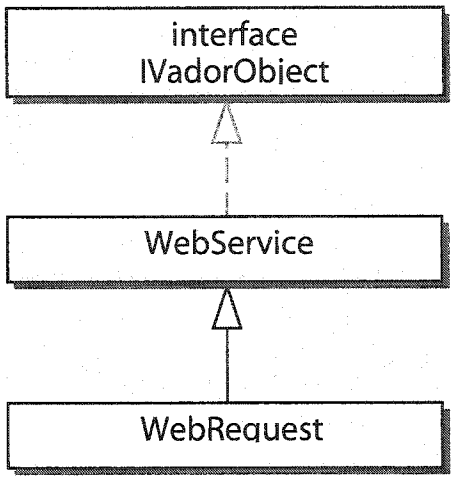

FIGURE 7.1 The WebService class diagram

information from the web browser. The WebService class can have other concrete sub-classes if the Vador Server provides other types of the web service. Figure 7.1 shows the WebService class diagram. This extension point comes from the Composite pattern.

- **WebVisitor – VadorVisitor**

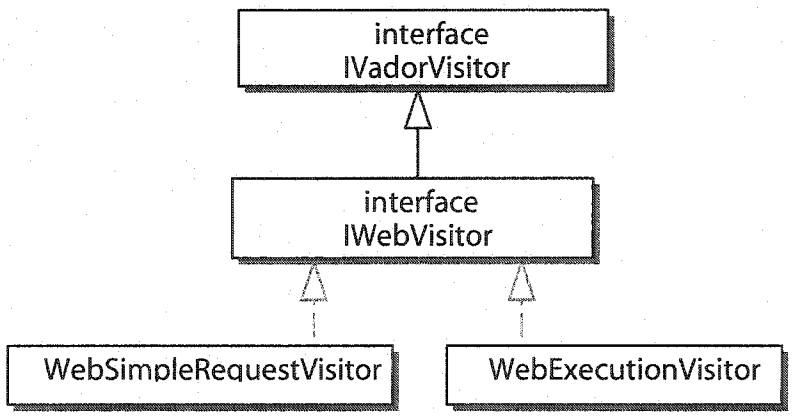

FIGURE 7.2 The WebVisitor classes diagram

The IWebVisitor interface extends the IVadorVisitor interface. It has several implementation classes which implement the different types of execution on the WebRequest objects. As shown in figure 7.2, the WebSimpleRequestVisitor performs the simple query tasks with the database and sends back the result to the web browser. The WebExecutionVisitor gets the DCInstance objects from the database, then asks the ExecuteDCInstance Agent (as so far, the ExecuteDCInstance Agent is under implementation, it will be a combination of the DCInstance object, the ExecuteSgyVisitor and the WrapperProxy) to execute DCInstance objects and sends back the execution results to the web browser. This extension point comes from the Visitor pattern.

- **WebServerProxy – Itinerary**

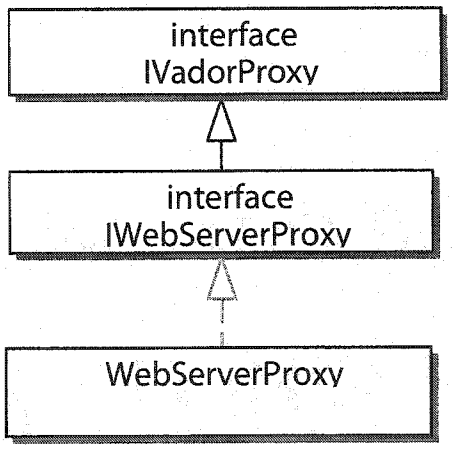

FIGURE 7.3 The WebServerProxy classes diagram

The IWebServerProxy interface extends the IVadorProxy interface, it has an implementation class – the WebServerProxy class which implements the mobility between the VadorWebServer and other servers. This extension point is located in the Proxy pattern. Figure 7.3 shows the WebServerProxy class diagram.

7.2.2.2 The VadorWebAgent module

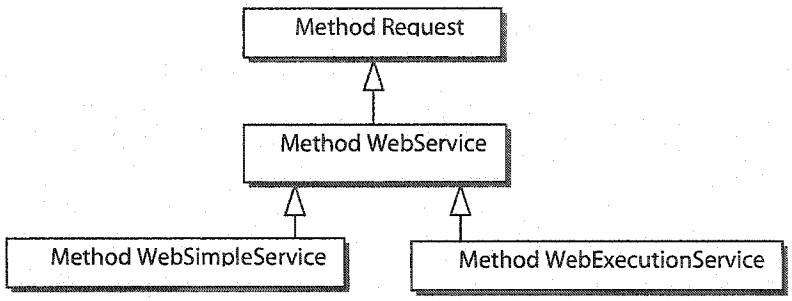

FIGURE 7.4 The VadorWebAgent class diagram

Figure 7.4 shows the VadorWebAgent class diagram. The Method_WebService abstract class extends the Method_Request class. This class defines the general behavior of the VadorWebAgent. As shown in figure 7.5, the VadorWebAgent has three components – WebService, WebVisitor and WebServerProxy.

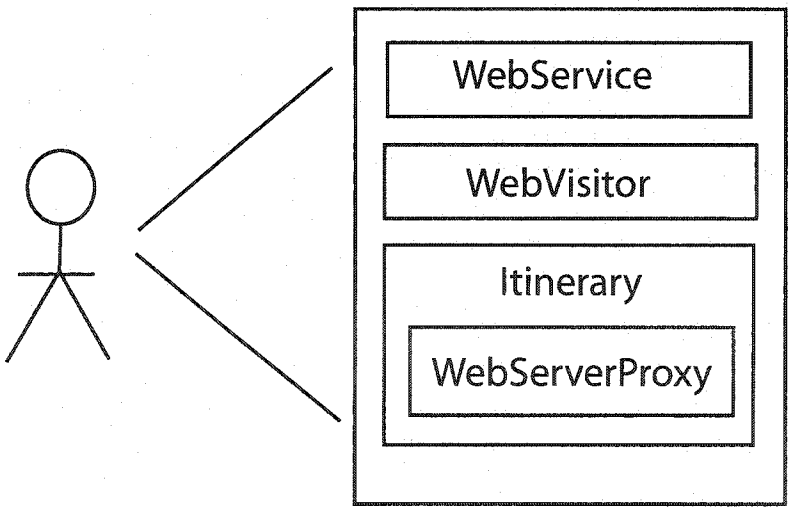

FIGURE 7.5 The components in the VadorWebAgent

The Method_WebSimpleService is a concrete sub-class of the Method_WebService

abstract class, it includes WebService, WebSimpleRequestVisitor and WebServerProxy objects (and possibly other proxies).

This class is a WebSimpleService Agent. It receives the requests from the web browser and keeps them in the WebService object. It uses the LibrarianProxy object to send itself to the LibrarianServer and uses the WebSimpleRequestVisitor object to query the database. It then uses the WebServerProxy object to come back on the VadorWebServer and uses the WebSimpleRequestVisitor to send back the results to the web browser.

The Method_WebExecutionService class is a second concrete sub-class of the Method_WebService abstract class. It includes WebService, WebExecutionVisitor and WebServerProxy objects.

This class is a WebExecutionService Agent. It receives the requests from the web browser and keeps them in the WebService object. It uses the LibrarianProxy object to send itself to the LibrarianServer and uses the WebExecutionVisitor to ask the OpenDCInstance Agent (as so far, the OpenDCInstance Agent is under implementation, it will be a combination of the DCInstance object, the DCInstanceBuilderVisitor and the LibrarianProxy) to get the DCInstance object from the database. It then ask the ExecuteDCInstance Agent to execute the DCInstance object uses the WebServerProxy object to come back on the VadorWebServer and uses the WebExecutionVisitor to send back the results to web browser.

7.3 Conclusions on this case study

Reflecting back on the introduction of the WebAgent service, we can report the following findings with respect to the overall VADOR architecture:

1. Vador System provides all the necessary extension points.

   Vador defines necessary extension points to get a new service in the Vador system.

2. The extension of Vador system is simple and clear.

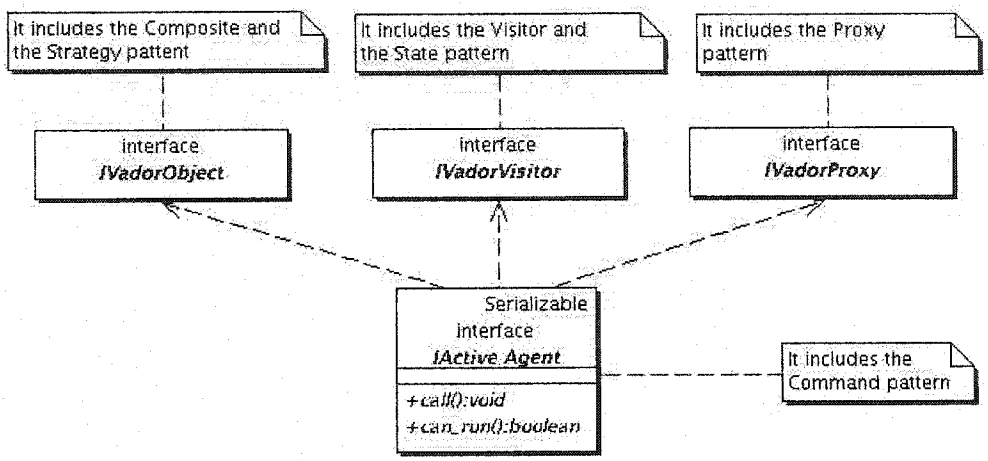

FIGURE 7.6 The Agent extension points diagram

Figure 7.6 shows the Agent extension points diagram. When a new service needs to be added in the Vador system, we should implement all these interfaces or use the default classes implemented by Vador.

3. The deployment of a new service is easy.

   When we deploying a new agent for a new service, the existing Vador Servers do not need to be restarted to use this new agent object as they know how to interact with the standard Active Agent interface.

CHAPTER 8

CONCLUSION AND FUTURE WORK

The Vador framework is a flexible and powerful software environment that provides a complete set of attributes in order to be effective in a real computational based design and analysis environment. It can be used to integrate and automatically execute complex analysis and optimization processes, and manage the results produced.

Key attributes of the Vador Framework are:

- Distributed computation,
- Manipulation of user data in its native format,
- Encapsulation of engineering applications,
- Graphical user interface,
- Database storage of meta-data.

The VADOR framework can be divided in three layers:

1. The presentation layer.

   We have the VadorGUI and the DBExplorer modules in this layer.

2. The application domain layer.

   We have the Librarian, the Executive, the Wrapper and the Wrapper_servlet servers, we also have the Analysis application in this layer.

3. The persistent data layer.

   We have the DataComponents Files and the DBMS.

There are two means of encapsulation in the Vador framework:

- *The DataComponents* encapsulates analysis results.
- *The StrategyComponents* encapsulates analysis applications.

We extensively use design patterns in the Vador system architecture. Design patterns describe how to establish communication between objects while hiding their data models and methods from each other. Keeping this separation has always been an objective of good object oriented programming. The use of design patterns in the design of the framework gives us a flexible and scalable architecture that specifically provides extension points to clients. This should allow incremental enhancements in functionality of the framework, while preserving robustness and maintainability.

The main design patterns in the Vador system are:

- The Mediator pattern, the Template Method pattern and the Adapter pattern which are located in the VadorGUI module.
- The Command pattern, the Composite pattern, the Strategy pattern, the Visitor pattern, the State pattern, the Proxy pattern and the Observer pattern which are located in the application domain layer.

The whole VADOR framework is designed using the Active Agent Pattern. The Agent in the VADOR framework works as behaves a mobile agent. A mobile

Agent is a small intelligent program that moves through a network automatically, searching for and interacting with services on the user's behalf. The Vador system uses specialized servers (the Librarian and the Executive server) that interpret the agent's behavior (opening objects, saving objects etc.) and communicate with other servers. A Mobile Agent (the opening strategy agent, the saving strategy agent etc.) has inherent navigational autonomy and can ask to be sent to some other nodes.

The Active Agent Pattern has six participants: client, user, agent, concrete agent, security manager and execution place. It implements the Agent Support System (ASS) itself, and is implemented on top of the Java Virtual Machine. Both server and clients run on top of the Java Virtual Machine (JVM). They can run in the same or different machines. Agents run on the Vador Servers and they interact with the end-user through the Vador application–VadorGUI.

The extension and deployment of the Vador system are simple and easy. The Active Agent Pattern provides all the necessary extension points to include new services, the main extension points in VADOR can be divided in three parts:

1. Extension Points in the VadorGUI Module.
    - New menu displaying status through the Mediator Pattern.
    - New DCViewers and StrategyViewers using the Template Method Pattern.
2. Extension points for other clients. Although we normally access the VADOR system through the VadorGUI application, we can also access it through other forms of client applications by using the Server's proxy class. For example, the Loadbalancing application (see Liu 2003) uses a batch client which directly accesses the Executive server through the Executive proxy class.

3. Extension Points in the domain layer.

    - New Agents based on the Command Pattern.
    - New composite objects using the Composite Pattern.
    - New Strategies using the Strategy Pattern.
    - New operations on Strategies using the Visitor and State patterns.
    - New proxy using the Proxy Pattern.

To create and execute a process in the Vador system, normally requires four steps:

- Define the DCType object.
- Define the StrategyComponent objects.
- Define the DCInstance objects.
- Execute the process.

Not only does Vador enable execution of the process, but it also provides the tools to help the management of the execution results and validation of processes. These tools include the Vador debug tool, the Vador Plot tool, the System monitor tool, a load balancing tool and an observer capacity.

The VADOR framework design structure is sound, but from the view point of performance and functionality, we can see that it has a number of shortcomings when compared with more general environments:

1. the communication system between the agents and servers is very simple,
2. the agent's anti-attack and recovery abilities are low,

3. the security policies/strategies are planned but not implemented yet.

Version 1.3 of the Vador Framework is currently under test and about to be deployed. We will continue to increase the functionality of Vador, make it more flexible, scalable and robust, and resolved some of the shortcoming. Some of these developments include:

1. implementation of an XML-based user-defined DataComponent, which will help the Vador system integrate user data into the Vador system, and allow operations such as display and editing of user data through the VadorGUI.

2. implementation of the Optimizer process.

3. use of **ssh** port forward technology instead of direct client/server communication mode in the communication system of the Vador Servers. This will increase the communication security in the Vador system.

4. use of **ssh** to log onto CPU servers on behalf of the user to execute a task instead of executing the task as the special user – "vador". Currently the "vador" user is a member of all groups that use the Vador system, the use of these groups involves granting some special directory group permission to allow the use of Vador. This restricts the use of Vador to secured environments, where Vador may be a member of all groups. The proposed modification will remove this requirement and thus increase execution security while using the Vador system.

REFERENCES

ALEXANDER, C., ISHIKAWA, S., SILVERSTEIN, M., JACOBSON, M., FIKSDAHL-KING, I., AND ANGEL, S. (1977). A pattern language. Technical report, Oxford University Press New York.

COOPER, J. W. (1998). *The Design patterns, Java Companion.* Addison-Wesley.

FAYAD, M. AND SCHMIDT, D. C. (1997). Object-oriented application frameworks. Technical report, Communications of the ACM.

GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J. (1994). *Design patterns.* Vuibert.

JOHNSON, R. AND FOOTE, B. (1988). Desihning reusable classes pattern. Technical report, Journal of Object-Oriented Programming. SIGS, 22-35.

KRASNER, G. AND S.T., P. (1988). A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Technical report, Journal of Object-Oriented Programming I(3).

LANGE, D. B. (1998a). Mobile objects and mobile agents: The aglet api. Technical report, General Magic, Inc.

LANGE, D. B. (1998b). Mobile objects and mobile agents: The future of distributed computing? Technical report, General Magic, Inc.

LIU, D. (2003). *Load balancing for distributed engineering applications.* Master thesis, École Polytechnique.

OBJECTSPACE (1997). Objectspace voyager corba integration technical overview. Technical report, ObjectSpace, Inc.

SCHMIDT, D., STAL, M. AND ROHNERT, HANS ANDBUSCHMANN, F. (1999). *Pattern-oriented software architecture.* John Wiley - Sons, Ltd.

SCHMIDT, D. C., JOHNSON, R. E. AND FAYAD, M. (1996). Software pattern. Technical report, Communications of the ACM.

SILVA, A. AND DELGADO, J. (2001). The agent pattern: A design pattern for dynamic and distributed applications. Technical report, INESC and IST Technical University of Lisbon.

SOBIESZCZANSKI-SOBIESKI, J. AND HAFTKA, R. (1996). Multidisciplinary aerospace design and optimization: survey of recent developments. *AIAA Paper 96-0711.*

SOBIESZCZANSKI-SOBIESKI, J. AND HAFTKA, R. (1997). Multidisciplinary aerospace design and optimization: survey of recent developments. *Structural Optimization*, 14, 1–23.

TRÉPANIER, J.-Y., LÉPINE, J. AND PÉPIN, F. (2000). An optimized geometric representation for wing profiles using NURBS. *Canadian Aeronautics and Space Journal*, 46, 12–19.

APPENDIX I

Vador Database UML diagram

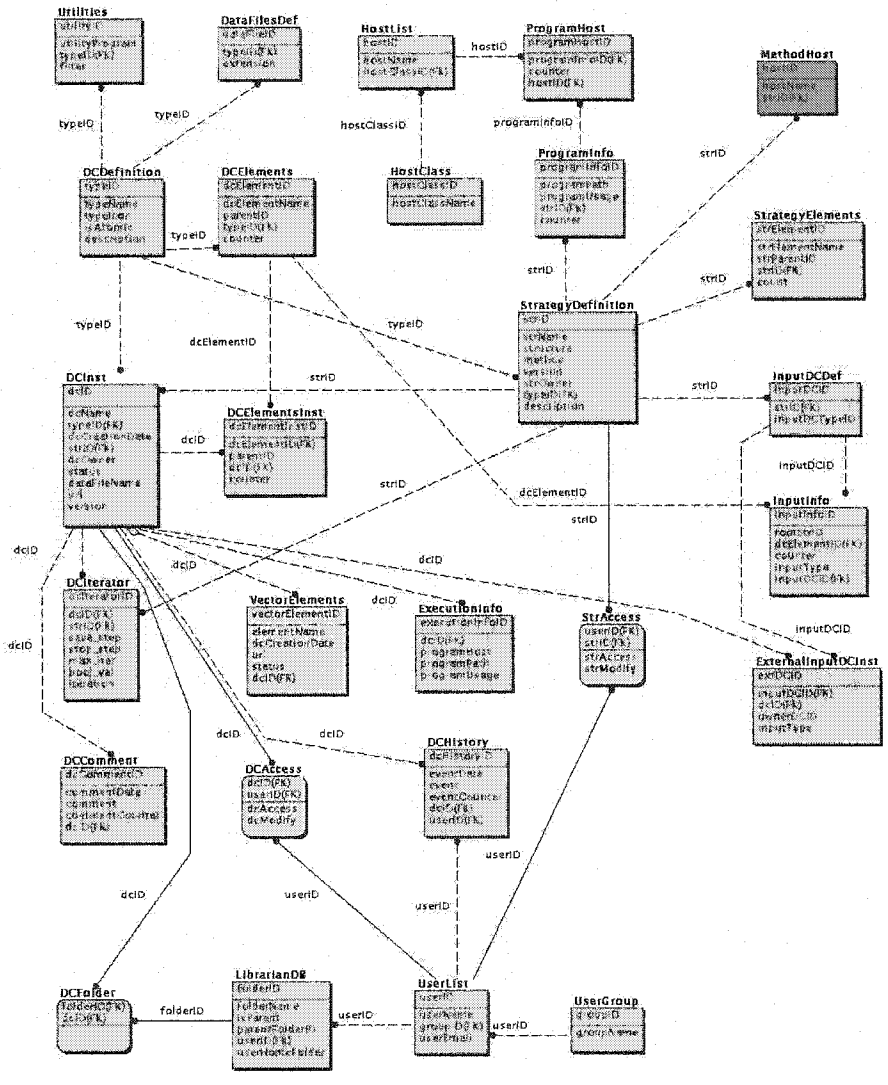

FIGURE I.1 Vador Database UML diagram

APPENDIX II

User Manual

This appendix presents a How-to document of the Vador framework.

II.1 Starting work with Vador

Before getting into a work session in the Vador System, the user should perform some preliminary work in order to translate his computation process in terms of Vador system components (File Types, Analysis Types, Files, Analysis, Programs, and Processes). The user is invited to identify clearly and individually all programs to be executed during a given process and all data files that will be produced and those that should be used as inputs. An easy way to do such a task is to draw a sketch of a generic flow chart of the whole process (fig II.1). Prepare data File's Types, input files locations, program's names, and their hosts. Once this is done, the user can launch the Vador GUI. Now let's use the Vador System to create the DTA process (fig II.1) and execute it.

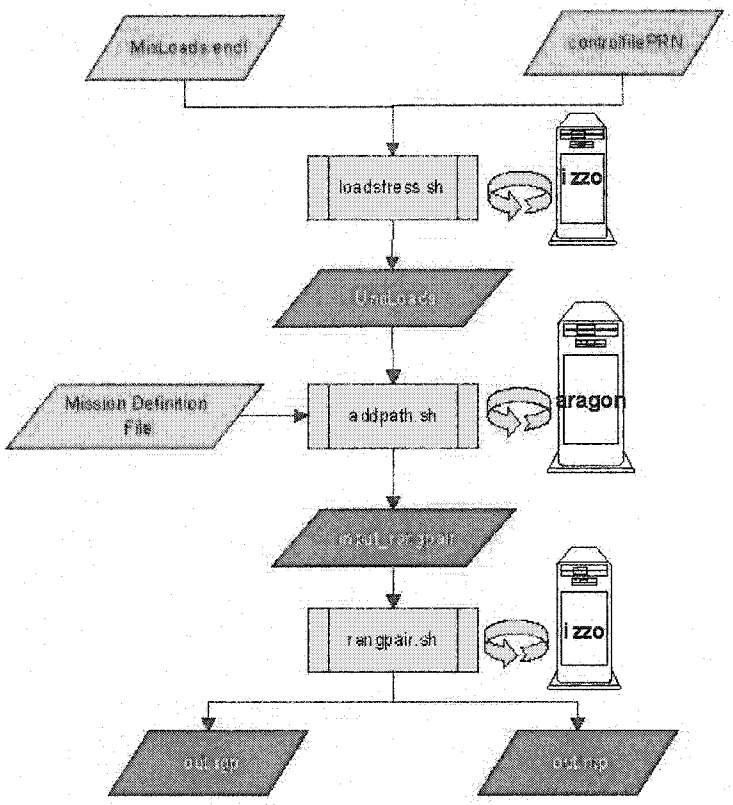

FIGURE II.1 DTA Process Flow Chart

II.2 Define a File Type (AtomicDCType) / Analysis Type (CompositeDCType)

II.2.1 A File Type

Start the "File Type Publisher" dialog box (fig. II.2) by clicking on "New File Type" button in the Tool bar menu. Fill in the text fields with the requested information:

1. in the first field enter the File Type Name, here is "MixLoads.endl";

2. in the second field, [choose a Type icon gif file] if you have already one, press the "Browse" button to locate your icon file in the system file. A default icon will be assigned by the Vador system if you don't provide one;

3. in the field "Description", type a textual description of the File Type (eg. what is intended for, which process it is part of, ...);

4. in the field "Extension" enter the encapsulated data file extension. If there is more than one extension, separate them by a comma ",";

5. in the field "Utility Program" enter the program name that can be used with the data encapsulated in the atomic Type as an editor or a visualization tool. If there is more than one utility program, use a "," to separate them;

6. press the "Save" to store new File Type information in the database or press the "Cancel" button at any time if you want to abort the definition process.

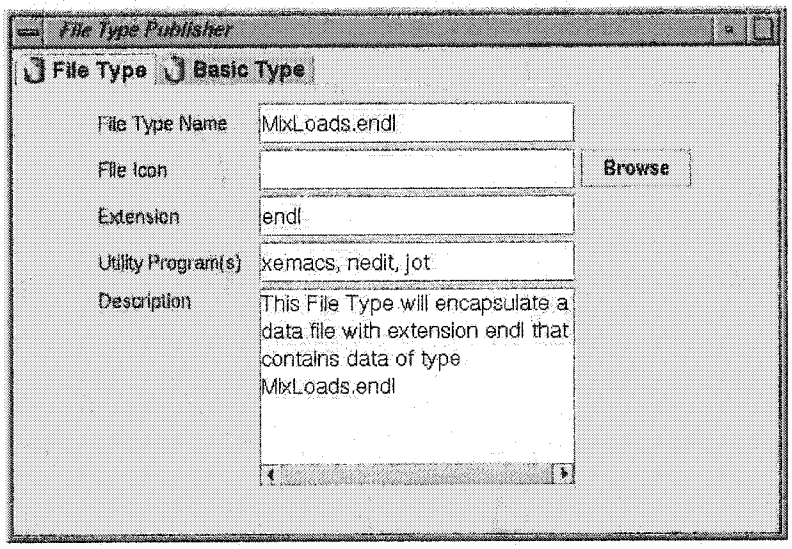

FIGURE II.2 File Type Publisher Window

II.2.2 An Analysis Type (CompositeDCType)

Once we have the File Types(AtomicDCTypes), we can combine them together to get an Analysis Type (CompositeDCType).

Suppose we have created the out_rgp.rgp File Type and the out_nrp.nrp File Type. We now want to use them to create the Analysis (CompositeDCType) whose name is RGP_Outputs.

Start the "Analysis Type Publisher" window (fig. II.3) by clicking on the "New Analysis Type" button in the Tool bar menu. The default type is a regular one, which means a collection of more than one File Type, and/or Analysis Type, unless you choose to build a Vector Type, which we will see in the next section. Fill in the text fields with the requested information:

1. enter in the first field the Analysis Type Name, here is "RGP_Outputs";
2. in the second field, [choose a Type icon gif file] if you have already one, press the "Browse" button to locate your icon file in the system file. A default icon will be assigned by the Vador system if you don't provide one;
3. in the field "Description" enter a descriptive text for the new composite DataComponent Type;
4. select each element that will be part of the new Analysis Type structure, one at the time, from the "Existing Types Classification" panel and press the "->" button (= Add the element to the new Analysis Type structure). The selected element will be added to the list of elements of the composition, in both middle and left panels. Repeat the same process for all elements. In case of mistake, you can remove any of the added elements by selecting it and

pressing the "<-" button (= remove from the list). Obviously, all elements, either single nodes or composite ones, of the composition are supposed to be already defined before this step in order to be included in the new composition. The user is recommended to proceed by defining leaves, using the "File Type Publisher", then intermediate composite nodes using "Analysis Type Publisher", and finish with the root of the hierarchical composition;

5. if you accept the new Analysis Type structure, you must press the "Save" button to store its information permanently in the database. At any time before saving, you can cancel the process;

6. if you choose to save the new Analysis Type into the database, and after all necessary transactions have been performed by the Vador System on the database, the new Analysis Type will be displayed on the right panel of the Vador GUI with the real attributes (especially typeID attribute that has been generated in database). (fig II.4).

II.2.3 A Vector Analysis Type (DCVector)

If we want to execute the RGP_Outputs Analysis type in a loop during the execution, we should define a Vector Analysis Type (DCVector) to encapsulate it.

Vector Analysis Type is another data model provided by the Vador framework. It is intended to be used for repetitive processes that perform several iterations. It's creation is straight forward as for a regular Analysis (composite) Type:

1. choose the "Vector Type" option by checking the radio button on the "Analysis Type Publisher", (fig II.5);

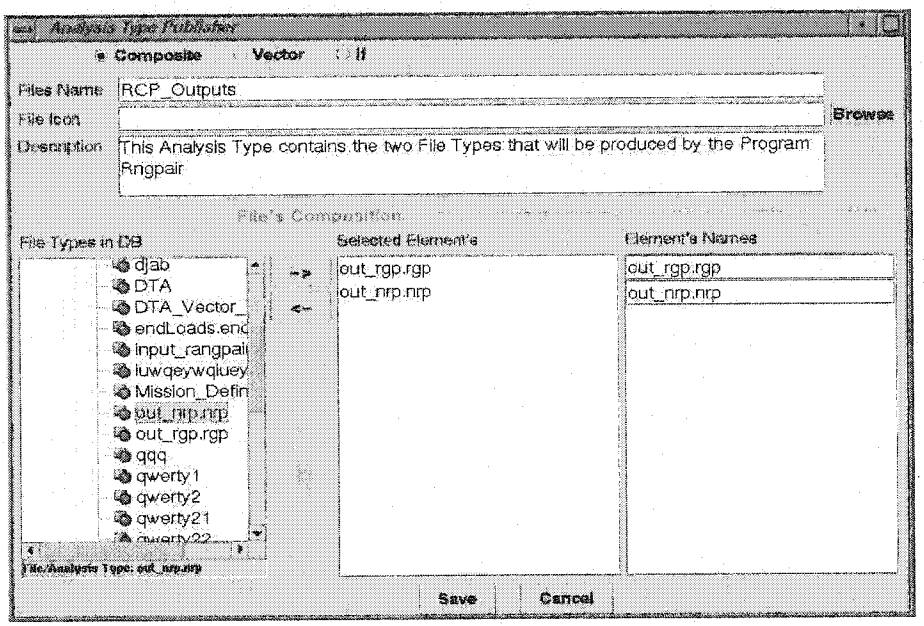

FIGURE II.3 Analysis Type Publisher Window

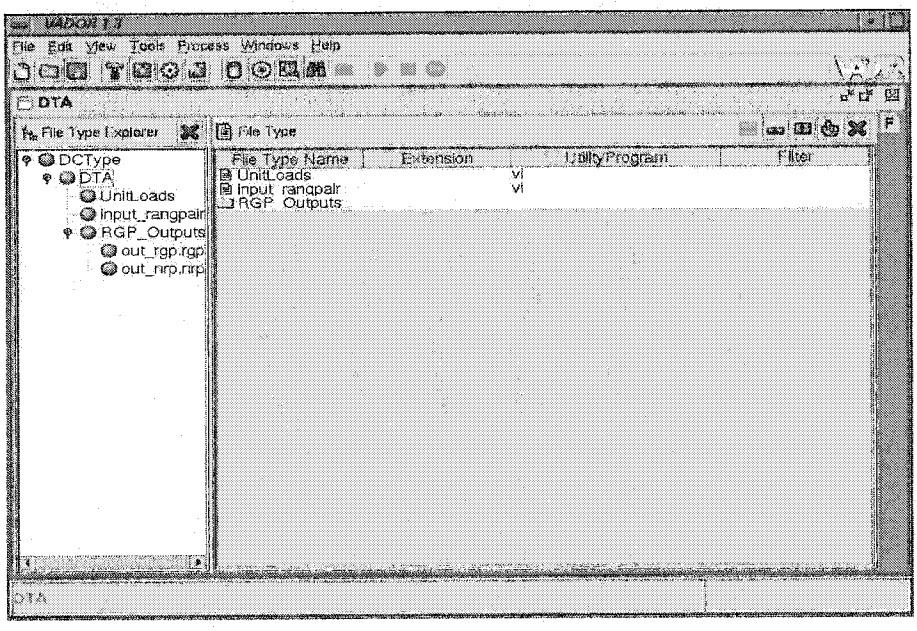

FIGURE II.4 New Analysis Type After it has been saved into the Vador database

2. do the same steps as for a regular Analysis (composite) Type definition; give it a name and a description;

3. select the element from the "Existing File/Analysis Types List" panel that defines the Vector elements type;

4. press the "Save" button if you want to save the new Vector Type in the database, or press 'the 'Cancel" button to cancel the operation.

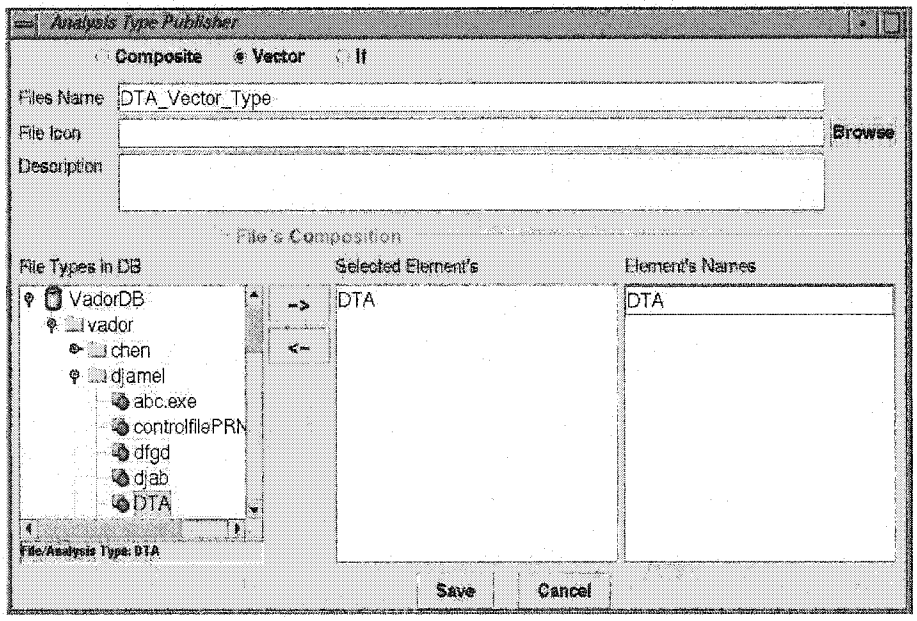

FIGURE II.5 Analysis Type Publisher Window: Vector Option

II.3 Define a New Program (AtomicSGY) / Process (CompositeSGY)

We have defined the Files/Analysis Type. Now let us define the Program (AtomicSGY)/Process (CompositeSGY) which will be used to create these Files/Analysis Type.

Vador Programs (AtomicSGY)/Processes (CompositeSGY) encapsulate user's programs and processes that are used to create data encapsulated in File/Analysis Types.

II.3.1 A New Program

A Vador Program encapsulates user's programs or scripts.

Suppose we want to define a Program (AtomicSGY). This Program will be used to create the UnitLoads File during the execution. To Define this Vador Program, do the following steps:

1. start the "Program Publisher" by clicking on the "Prog. Publisher" sub-menu from the "Tools" menu or the "New Program" from the Vador GUI tool bar menu. A dialog box as in figure II.6 will appear;

2. type the Vador Program name in the field "Program Name", here is Load-Stress;

3. select the output File/Analysis Type by pressing the "Browse Type" button in front of the "Output File(s)" field. The classification of all File/Analysis Types existing currently in the database will be displayed (fig. II.7). Select the output File/Analysis that will be created by the new Vador Program, and press the "Select" button. The selected File/Analysis Type name will be displayed in the "Output File(s)" field. Here we select UnitLoads File;

4. press the "Configure" button in front of the "Executable" label to select the set of hosts, the user program paths, and the user program name ;

5. Press the "Configure" button in front of the "Writers" label to select the list of the people that can modify the current new Vador Program;

6. Press the "Configure" button in front of "Readers" label to select the list of the people that can access the current new Vador Program;

7. specify the exact number of arguments, in the same way and order as they are requested by the encapsulated user program in the field "Number of arguments", and press the "Configure" button. A new dialog box will appear. It contains setting rows in the specified number of arguments with the options: "IN", "OUT", "INOUT", and "FLAG". The user should proceed here carefully by respecting his own program arguments order;

8. you can inspect the new Program usage expression by pressing the "View" button which becomes enabled only if the usage has been completed;

9. Press the "Save" button to save program arguments in the database.

II.3.2 A Conditional Program (ConditionalSGY)

It is a special kind of Vador Program which creates a Boolean File Type that can hold a boolean value (true/false). To create a Conditional Program follow the same steps as for a regular Vador Program. Select the "Conditional" radio button in the "Program Publisher". The default output will be set by Vador System to be a "Boolean Type". For the remaining Program infos, do the same steps as for a regular Vador Program.

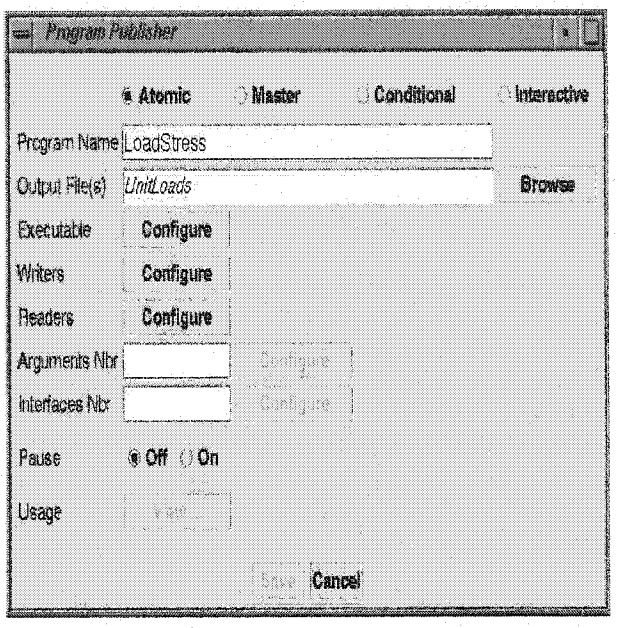

FIGURE II.6 Vador Program Publisher Window

II.3.3 A New Process (CompositeSGY)

As stated before, Vador Processes are user-defined processes using structured procedural programming concepts: sequential blocks, parallel blocks, if statements, and controlled iterations. We will introduce each type of these Processes in the following sections. The major difference between the different process types resides in the execution of each of them.

In order to build a new Vador Process, one should first create all its elements: Programs , and Processes, if there is any, that are part of the new Vador Process structure. If this has been already done, the next step is to open the "Process Publisher" window (fig II.8), either from the "Tools" menu, or using the "Process Publisher" button icon in the tool bar menu on Vador GUI.

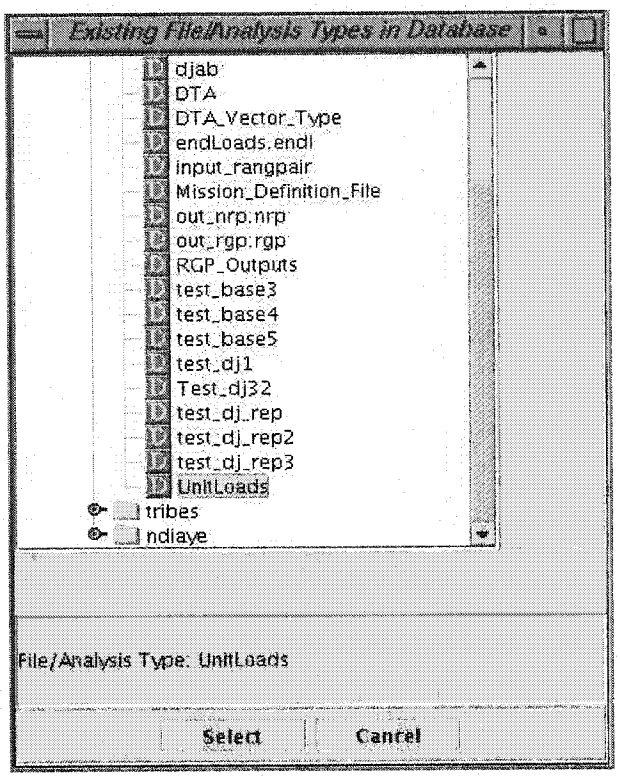

FIGURE II.7 Vador Program Publisher: Output Type Selection

Then open the Analysis Type that will be created by the intended new Vador Process. If the user selects any kind of new Process structure (Sequential, Parallel, IF, DoWhile, ...) without opening an Analysis Type, an warning message will be displayed: "Please click the new Process button to select an Analysis Type first".

Once an Analysis Type has been selected, it is opened on the "Process Publisher" up panel. The approach to create a new Vador Process is almost the same for all kind of process's structures. However, depending on the type of Vador Process, some detailed steps may be different.

Here, let's create the Vador Process which will be used to create the DTA Analysis Type during the execution.

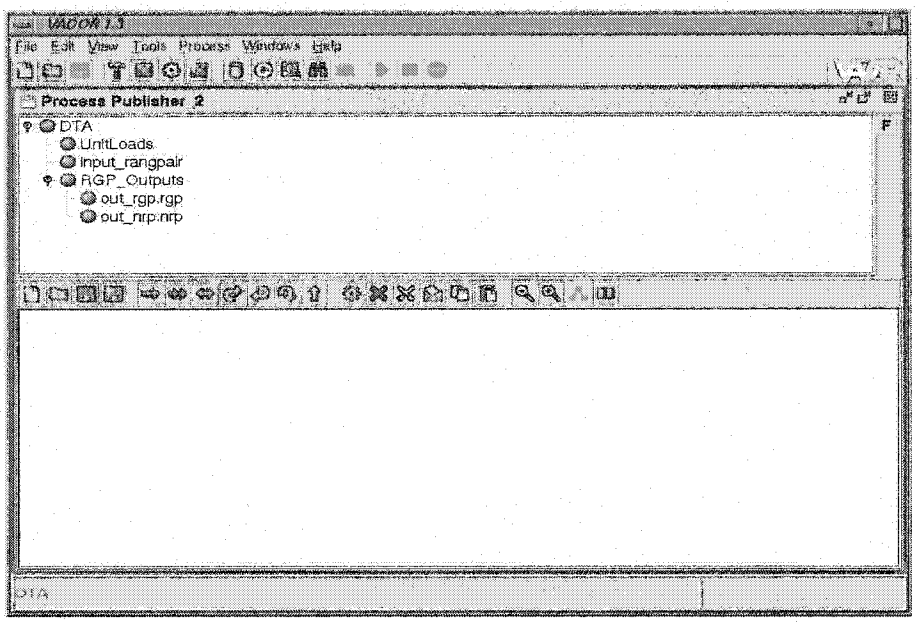

FIGURE II.8 Vador Process Publisher Window

1. **Sequential Process**

   Assuming that the user has defined individual Vador Programs that are elements of the Process "DTA_Process", namely:

   - "loadStress" that creates the File type "unitLoads",
   - "addPath" that creates the File type "inputRGP",
   - "rangPair" that creates the Analysis Type "RGPoutputs".

   (a) press the "Sequential" button of the vertical menu bar in the "Process Publisher" window. An input dialog box will be displayed prompting the user to enter the new Process name and the number of the elements in the new composite structure. The default value is 2;

   (b) after entering the name, and the children number ("DTA_Process" string, and the number 3 for DTA process) (fig II.9), press the "Ok"

button, a graphical representation of the new Process will be displayed in the bottom panel of the "Process Publisher". See Figure II.10;

(c) an important step in the new Vador Process definition is to establish the link between the new Process elements and the Analysis Type elements, one by one, starting from the Analysis Type tree node with the Process element. This can be achieved by clicking first on the process element, then clicking on the Analysis Type node, and finally validate the relationship by clicking on the "Create" button. Such action is the way to indicate to the Vador System which process will create an Analysis Type;

(d) repeat the same procedure for each of the Process and Analysis Type. Because the Process elements have been already defined and stored in the database, when the user makes the link between a child Program/Process and the corresponding element in the Analysis Type tree, the list of Programs/Processes that can create the selected Analysis element Type will be searched from the database and displayed on a dialog box (fig II.10). Select from the list the Program/Process you want to associate to the Analysis element Type and press the "Use" button. The corresponding Program/Process rectangle will be labeled with the selected Program/Process name, and the element File/Analysis Type to create;

(e) if the selected Program/Process needs some File Type as input(s), the user should proceed to input setting (see sub-section 6 **Input Setting**). A red button labeled "Input Files", will be added to the Program rectangle. It will be used to identify the real input Files, as we will see later;

(f) After making all necessary links between the new Vador Process struc-

ture and Analysis Type tree (fig. II.11), press the "Save" button (identified by a "floppy disk" icon) on the menu-bar of the "Process Publisher". The new Sequential Process information will be saved to the database and will be available to be used to build Analysis instances of "DTA_Type" Type.

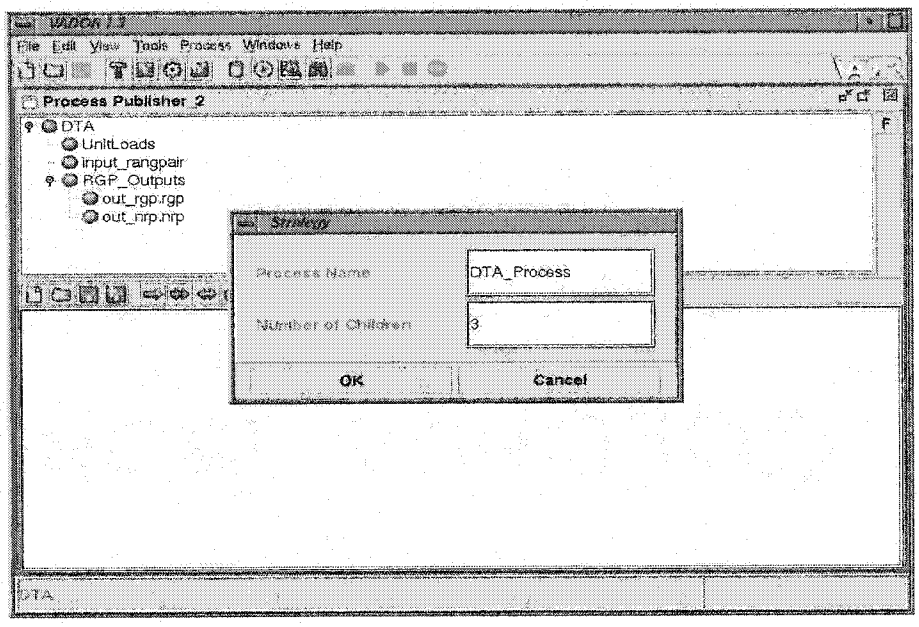

FIGURE II.9 Vador Sequential Process

2. **Parallel Process**

The creation of a Parallel Process is quite similar to a Sequential one, except in some details, as for example the graphical representation of the Parallel Process children is different with that of the Sequential Process (fig. II.12).

3. **If Process**

(a) press "if" button in the bottom menu of the "Process Publisher" window, and enter the name of the new "If" Process. You will see a if graphical

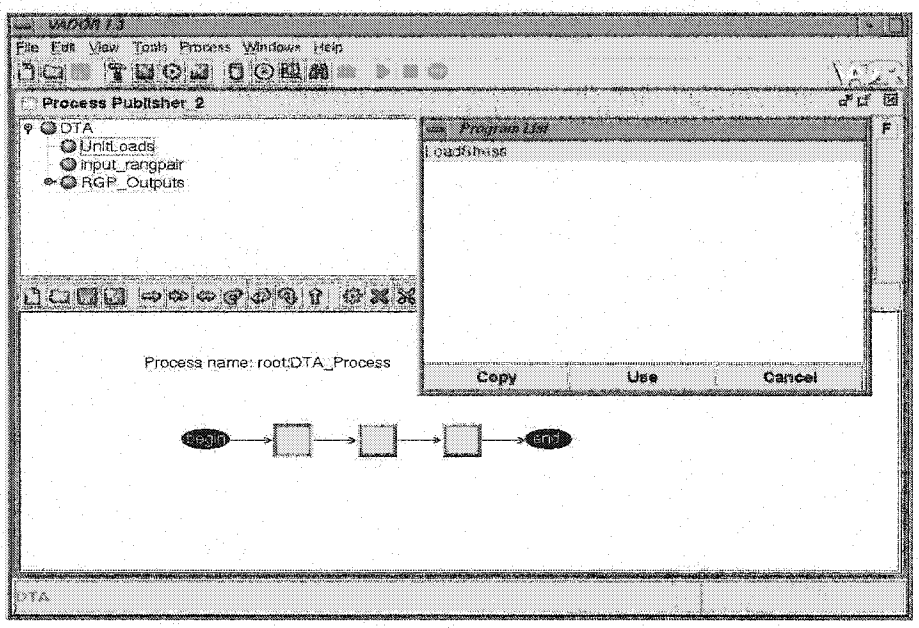

FIGURE II.10 Child Program/Process Selection corresponding to the Chosen File Element Type

representation;

(b) establish all the necessary links between the Process elements and the Analysis Type tree elements, starting from the root Process, as for the Sequential Process. Then, press the "Save" button to store the new "If" Process information in the Vador database. (fig. II.13).

4. While Process

A While Process is a composite one with a special structure, and an output of Vector Type. The Process tree contains in the first level a "Conditional Program" node, and a "Body Process" node which can be a Program (single) or a whole Process (a composite one). Its particularities arises from the way it will be executed. Its "Body Process" will be executed iteratively depending on the "Conditional Program" result "Boolean File". Lets consider, for

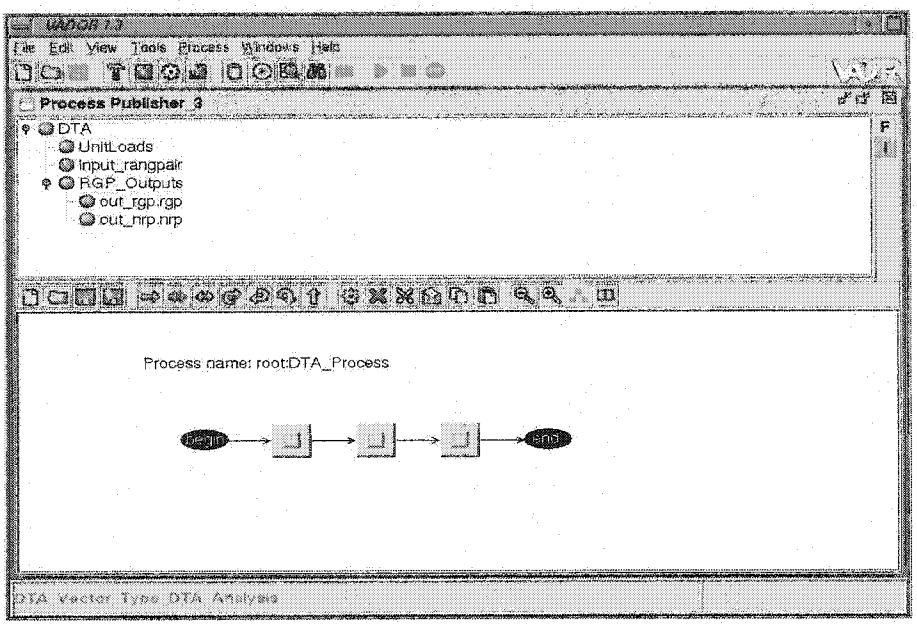

FIGURE II.11 Final View of a Sequential Vador Process Definition

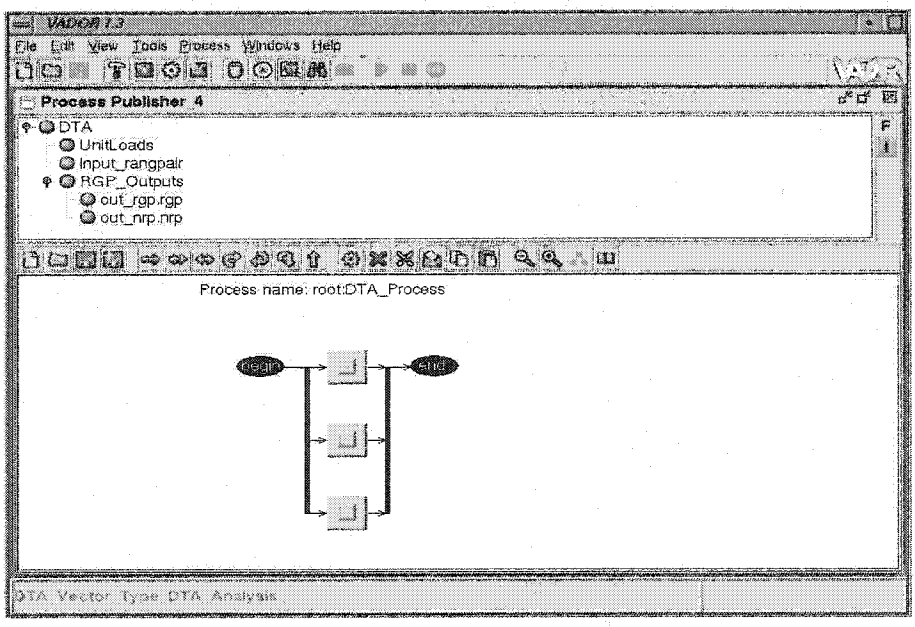

FIGURE II.12 An Example of Parallel Process Definition

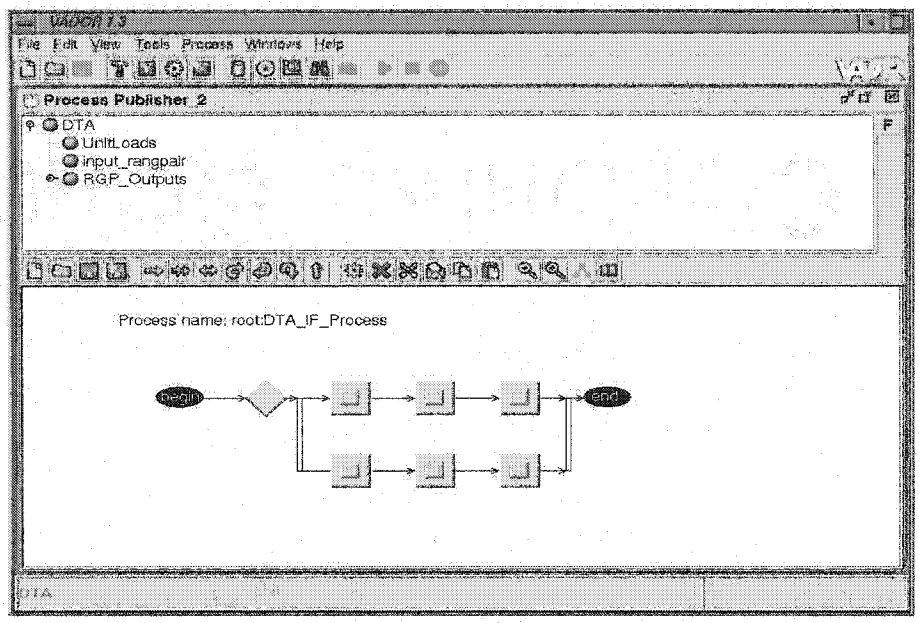

FIGURE II.13 An Example of If Process Definition

explanation purpose, the definition of a "While Process" for the Vector Type "loopDTA" which contains as a "Body Process" the "DTA_Type" introduced in the previous sections:

(a) press the "New Process " button from the Vador GUI menu-bar that launches the "Process Publisher" frame;

(b) open the Vector Type "loopDTA" for which you want to define a "While Process";

(c) press the "While" button on the menu-bar of the "Process Publisher" frame. An input dialog box will be displayed prompting the user to enter the Process Name. Enter the Process name and press the "Ok" button, the new "While Process" graphical representation will appear on the bottom panel (fig. II.14);

(d) make the links between the Vador While Process elements and the Anal-

ysis Type tree, For the case of the "Conditional Program", there is no corresponding node in Analysis Type tree. All you have to do is to select the graphical "Conditional Program" box on the "Process Publisher" window, and then press the "Create" button, a dialog box will appear with the list of "Conditional Programs" existing in the Vador database. Select the one you want to use for the current new "While Process" and then press the "Ok" button;

(e) do the same exercise with the "Body Process" in order to set the output File/Analysis Types. For "loopDTA" example, the Body Process will be one of the Processes that have been built for the "DTA_Type";

(f) press the "Save" button to store the new "While Process" attribute's values in the Vador database.

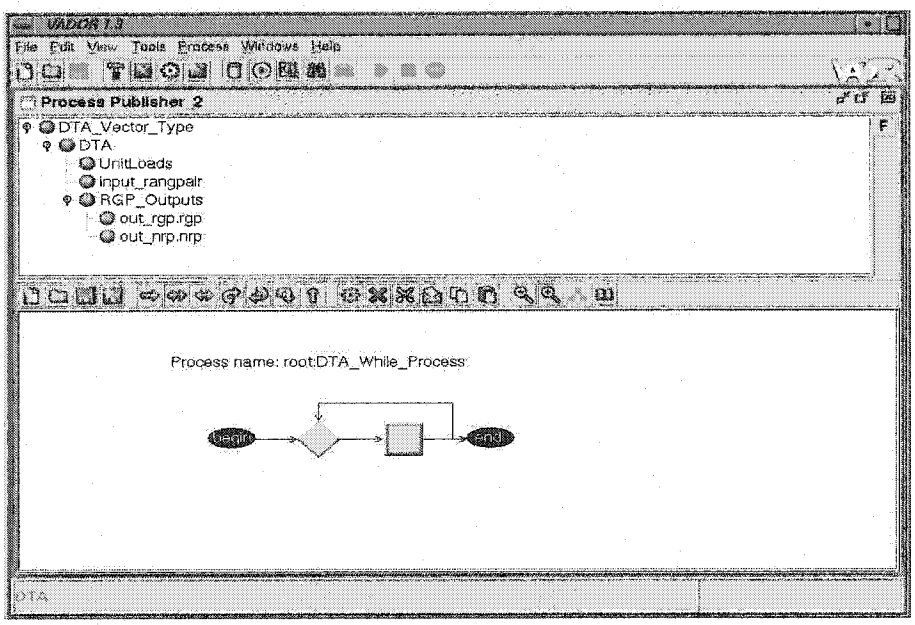

FIGURE II.14 Vador While Process Definition's resulting Structure

5. **DoWhile Process**

Vador DoWhile Process definition is exactly similar to the While Process definition described in the previous section. The only difference is in the Process execution. During DoWhile Process execution, the first iteration is always done, then the loop condition (result of the "Conditional Program" execution) will be tested in order to decide to make the next pass or not. For the While Process, the loop condition is tested before executing any iteration.

6. **Input Setting**

A Vador Process is a collection of other Vador Processes and/or Vador Programs. Once the user has created a new one, he should proceed to the input identification, if there is any, in individual Vador Programs that are part of the new Process structure. Remember that all inputs to a Program that is not yet part of a the composite Process are obviously "External". Once the same Program becomes part of a new Vador Process, one should specify if the inputs remains "External" (do not come from the produced Analysis) or becomes "Internal" or "Iterative" ones. When the user selects a Program box in the "Process Publisher" and indicates the Analysis Type that it will build by pressing the "Create" button, the list of Programs that create the selected File/Analysis Type will be displayed (fig. II.10). If he chooses a Program (individual node structure), a set input panel will be displayed (fig. II.15) on the upper part. It has four panels: the second panel displays the list of input File Types for the selected Program. Proceed as follow:

(a) select an input File Type from the second panel of the set input panel. Then specify if it is "External", "Internal" (it can be iterative only if the Process is an iterative one: "For" Process, "While" Process, or "DoWhile" Process, otherwise the "Iterative" option will not appear on the set input

panel);

(b) press the "»" button. The selected type name will moved to the third panel;

(c) repeat the previous steps for the remaining unset types, We should notice that if an input is set to be "Internal" but it does not appear in the Analysis Type tree (top left panel), then an error message will be displayed for that purpose.

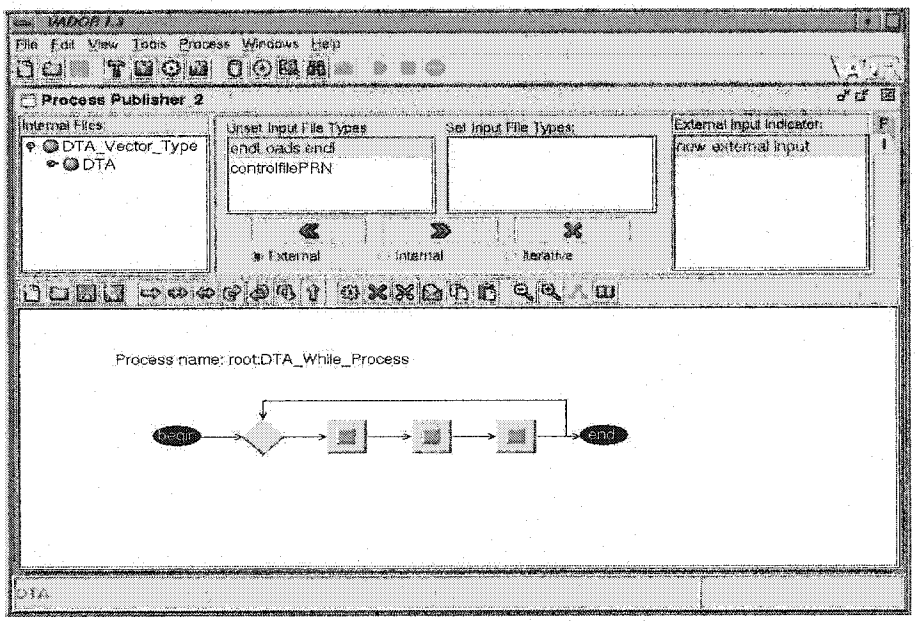

FIGURE II.15 Input Setting Window

II.4 Create a New Analysis (DCInstance)

1. First, create a File/Analysis Type, and a Program/Process that creates that Type;

2. press the "New Analysis" button of the menu-bar, or the "New Analysis" sub-menu of the "File" menu. A dialog box entitled "New Analysis" will appear. It shows the classification tree of all File/Analysis Types that already exist in the Vador database;

3. select the File/Analysis Type you want and press the "Next" button. The list of Programs/Processes that can create the selected File/Analysis Type will be displayed;

4. select the Program/Process that you want to use for the new Analysis creation, and press the "Next" button;

5. in the text field "File Name", enter the string <name> of the new File/Analysis. Here we enter DTA_Vector_Type_DTA_Analysis;

6. enter the [list of people] that you allow to access your new File/Analysis in the field "Readers". It is optional in the current version;

7. enter the [list of people] that you allow to modify your new File/Analysis in the field "Writers". It is optional in the current version;

8. press the "Next" button, and select the folder where you want to store the new File/Analysis in the Vador Classification System;

9. press the "Save" button to store the new File/Analysis information in the Vador database. On the completion of the previous steps, the structure of the new File/Analysis will be displayed on the Vador main window;

10. The graphical representation of the Program/Process structure will be automatically displayed on the bottom half of the Vador GUI (fig II.16);

11. complete the "External" inputs setting if there is any (in fact, the "Internal", and "Iterative" input(s) has been already defined when the Program/Process

is created in the "Program Publisher" or "Process Publisher". Click on the "Input File" red button appearing on each Program box, a set input panel appear with four panels (fig II.17). The fourth panel ("Unsetted input type") displays the list of "External" input Files that are of the same Type as the one specified for the input(s). Each time an input File Type is selected on the second panel ("Unsetted Input"), the list of File(s) of the same Type in database will be displayed on the fourth panel. Select the input File Type and the input File name, and press the "»" button. The File name will be added to the third panel. Repeat the same action for all elements of the second panel. After setting the "Input File", the button(s) appearing on the Program(s) boxes on Vador GUI will change color to green. The first panel displays the current Analysis tree (from which come the "Internal" input(s)). At any time, the user can check which File(s) are used exactly as input by pressing "Input File" button for any Program;

12. do the same actions for all inputs and for each Vador Program that is part of the whole Vador Process associated to the current Analysis. Press the "Save" button of the Vador GUI menu-bar to save permanently in the database the input files defined.

II.5 Execute a File/Analysis

Executing a File/Analysis in Vador System comes to execute its associated Program/Process.

1. load the File/Analysis you want to execute;

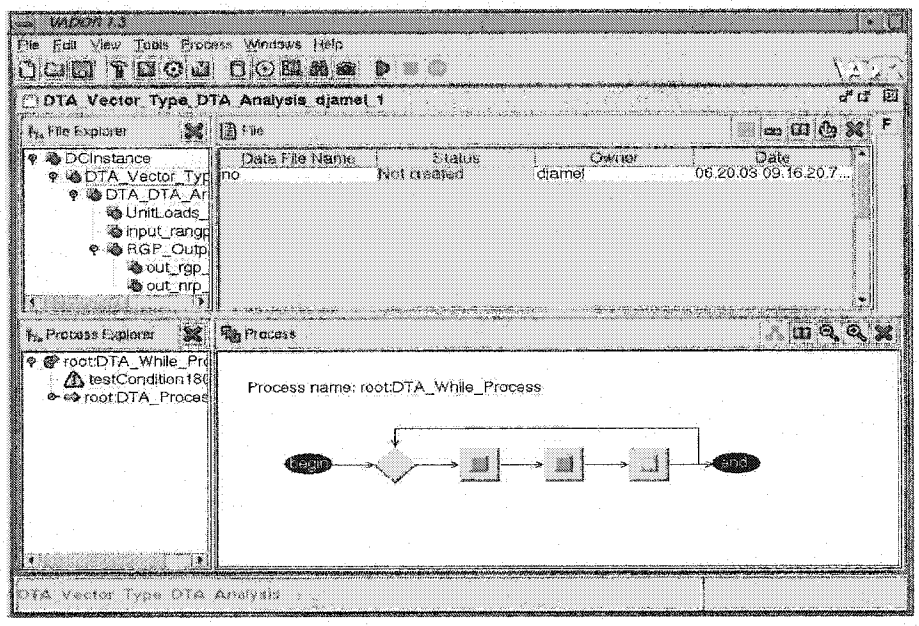

FIGURE II.16 New Analysis After it has been saved into the Vador database

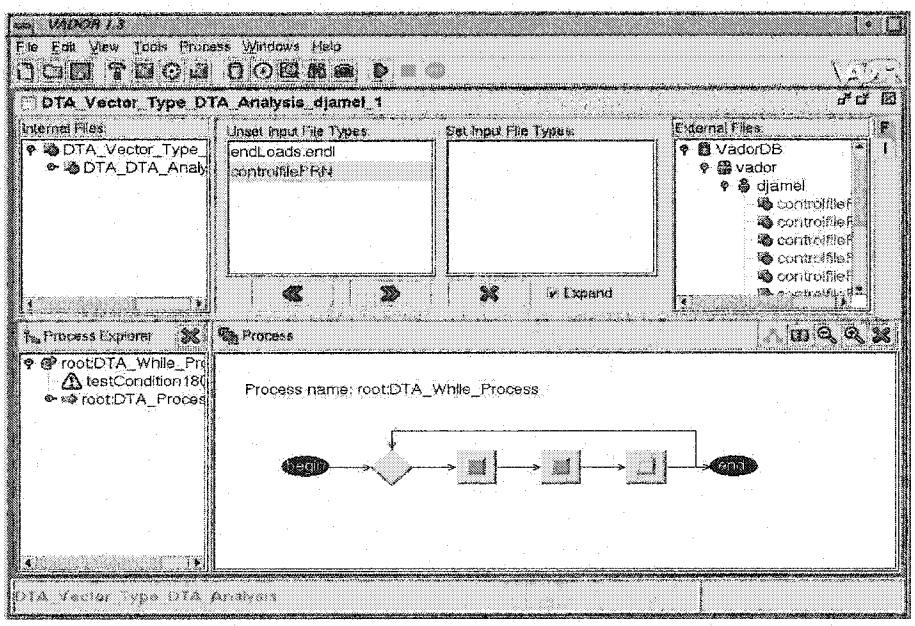

FIGURE II.17 Input Setting Dialog Box

2. check the inputs definition. If they have not been done before, introduce each of them and press the "Save" button;

3. press the "Execute" button of the Vador GUI bar-menu. A dialog box entitled "Execution Management" will be displayed. It gives the user two options to choose the Program/Process execution time. The user can decide either to execute his Program/Process immediately, or delay to another time. For the last option, the user should specify the date and hour for starting the execution by the Vador Servers. When the execution is launched from the Vador GUI, only the File/Analysis identifier "dcID", and the "User Name" will be transmitted from the Vador GUI to the "Executive Server" which will coordinate the File/Analysis execution. It should first request the File/Analysis object information from the "Librarian Server", extract the associated Program/Process, and request the wrapped user's programs execution on the "CPU Server" side. If everything goes well, the File/Analysis graphical representation elements on the GUI will be colored to green. In case of problems, message errors will send back to the user. Also, in any case, the execution information can be viewed on the Vador GUI. For that purpose, press the "Debug" button to check execution steps detailed information. It will be displayed in tree format with the same structure for the whole Program/Process on the right panel of the bottom half of the Vador GUI. When this tree is expanded, each Program in the Process tree will have three nodes called "prexecution", "execution", and "postexection". If you click on any of these Program nodes, a text format will be displayed in the right panel of the bottom half of the Vador GUI. There are two major steps for Vador Program execution: "BEGIN" execution, and "END" execution. It gives some important details related to the Program execution by the "CPUServer" such the "Host" Name, the "Time" of execution, the user's program "Path", and all its

arguments such as inputs and outputs "Path".